Erweiterung und Optimierung einer Kommunikationsplattform für PC-basierte Steuerungssysteme

Von der Fakultät Elektrotechnik, Informationstechnik, Physik

der Technischen Universität Carolo-Wilhelmina

zu Braunschweig

zur Erlangung der Würde

eines Doktor-Ingenieurs (Dr.-Ing.)

genehmigte Dissertation

von:	Dipl.-Ing. Yannick Dadji Foyet
aus (Geburtsort):	Yaounde (Kamerun)
eingereicht am:	27. Juni 2011
mündliche Prüfung am:	25.Januar 2012
Berichterstatter:	Prof. Dr.-Ing. Harald Michalik Prof. a. D. Dr.-Ing. Jörn-Uwe Varchmin
Vorsitzender:	Prof. Dr.-Ing. Walter Schumacher
Druckjahr:	2012

Danksagung

Die vorliegende Arbeit ist das Ergebnis meiner Tätigkeiten als wissenschaftlicher Mitarbeiter am Institut für Elektrische Messtechnik und Grundlage der Elektrotechnik (emg) und am Institut für Datentechnik und Kommunikationsnetze (IDA) an der Technische Universität Braunschweig. Mit großer Dankbarkeit blicke ich auf diese letzten fünf Jahre zurück, wo ich das Teilprojekt B1 „Steuerungs- und Kommunikationsarchitekturen“ des Sonderforschungsbereichs (SFB) 562 „Robotersysteme für Handhabung und Montage“ bearbeiten durfte.

Allen voran möchte ich Gott, dem Herrn, danken, dass Er mir Leben und Verstand geschenkt und den Weg für ein Studium und eine Promotion in Deutschland geebnet hat. Ich danke ebenso meiner lieben Frau, Huguette Djoumsap Takam, für ihre ausgeprägte psychologische und praktische Unterstützung, sowie für ihre Geduld, Verständnis und Motivation während der Zeiten besonderer Belastung.

Für die Ausbildung und das erworbene Fachwissen bin ich Deutschland und der TU Braunschweig dankbar. Insbesondere danke ich Herrn Professor Jörn-Uwe Varchmin für sein langjähriges Engagement für die Integration von ausländischen Studierenden, seinen hilfreichen Beistand während der Studienzeit und dafür, dass er an meine Fähigkeiten geglaubt und mich ermutigt hat, den Promotionsweg zu gehen. Meinem Doktorvater, Herrn Professor Harald Michalik, danke ich für die fachliche Betreuung und die stets konstruktiven Beiträgen in unseren Arbeitsgruppen. Dankbar bin ich auch meinem fachlichen Vorgänger Dr. Nnamdi Kohn, dessen strukturierte Arbeitsweise mir eine erfolgreiche Einarbeitung in das komplexe Themengebiet der PC-basierten Steuerungs- und Kommunikationsarchitekturen ermöglicht hat. Meinem ehemaligen Kollegen, Herr Tobias Möglich, danke ich für die fruchtbare Zusammenarbeit besonders bei der Entwicklung der FPGA-basierten Kommunikationsknoten, sowie für die ermutigenden fachlichen und persönlichen Gespräche. Den Herren Björn Osterloh, Dietmar Walter, Frank Bubenhagen und Dr. Torsten Fichna danke ich für deren Unterstützung bei der Entwicklung der FPGA-Knoten.

Zuletzt einen besonderen Dank an meinen SFB-Kollegen aus unterschiedlichen Fachgebieten für das angenehme Klima bei den Projektgesprächen, für die zahlreichen Hinweisen, Anmerkungen und Kritikpunkte, die einen Beitrag zum Entwurf und zur Umsetzung der in dieser Arbeit vorgestellten Konzepte beigetragen haben. Dabei möchte ich besonders die Herren Matthias Bruhn, Dr. Jochen Maaß, Dr. Thomas Reisinger, Dr. Frank Wobbe, Jens Steiner, Dr. Christoph Stachera und Dr. Michael Kolbus erwähnen.

Inhaltsverzeichnis

1 Einleitung

Der Robotikmarkt ist eine äußerst dynamische Wachstumsbranche, die in den letzten Jahren einen enormen Entwicklungsprozess durchlaufen hat. Allein in Deutschland ließ sich im vergangenen Jahrzehnt ein kontinuierliches Wachstum des operierenden Bestands von Industrierobotern registrieren (Abbildung 1-1). Hinsichtlich der technologischen Entwicklung ist auch eine zunehmende Funktionalität der Roboter zu verzeichnen. Roboter entwickeln sich immer mehr zu einem universellen Kernelement der Automatisierung, welches aus Maschine, Software sowie Dienstleistungen besteht [VDM10]. In der Industrie werden derzeit überwiegend serielle Roboter eingesetzt. Strukturell bestehen sie aus einer offenen kinematischen Kette, bei der alle Glieder und Gelenke hintereinander angeordnet sind und jedes Glied über je ein angetriebenes Gelenk mit seinen benachbarten Gliedern verbunden ist. Aufgrund ihres strukturellen Aufbaus weisen serielle Roboter einen großen Arbeitsraum und eine hohe Beweglichkeit des Manipulators auf. Allerdings stehen diesen Vorteilen einige strukturbedingte Nachteile gegenüber; aufgrund der bewegten Massen lassen sich keine hohen Geschwindigkeiten und Beschleunigungen erreichen, was im Bereich der schnellen Handhabung zu Einschränkungen der industriellen Produktivität führt. Außerdem grenzen die geringe Steifigkeit der Struktur und die wegen des sich kumulativ fortpflanzenden Gelenk- und Getriebespiels begrenzte Positioniergenauigkeit das Anwendungsspektrum dieser Roboter im Bereich der Montage ein.

Anders als serielle Roboter bestehen parallele Roboter aus einer geschlossenen kinematischen Kette. Sie zeichnen sich dadurch aus, dass der Manipulator über mehrere Führungsketten mit dem Gestell verbunden ist [Mer97]. Ein wesentlicher Vorteil von parallelen Strukturen ist die Möglichkeit, alle Achsantriebe am Gestell fest zu montieren. Aufgrund der daraus resultierenden geringen zu bewegenden Massen pro Antrieb ergeben sich sehr günstige dynamische Eigenschaften, die hohe Arbeitsgeschwindigkeiten und -beschleunigungen ermöglichen. Gleichzeitig lässt sich durch den strukturellen Aufbau eine höhere Genauigkeit und Steifigkeit erreichen. Die angewandte Steuerungstechnik muss allerdings Konzepte integrieren, die parallelroboterspezifische Besonderheiten berücksichtigen. Zu diesen Besonderheiten zählen beispielsweise die Möglichkeit der Kollision von Strukturelementen untereinander, die Singularitäten und zuletzt die variable lastabhängige Trägheit des Manipulators, die bei hohen Verfahrgeschwindigkeiten identifiziert und berücksichtigt werden muss.

Innerhalb des Sonderforschungsbereichs 562 „Robotersysteme für Handhabung und Montage“ [SFB10] der Deutschen Forschungsgemeinschaft wurde der Einsatz von Parallelrobotern für die Handhabung und Montage erforscht. In diesem Zusammenhang wurde im Rahmen der Arbeit „Kommunikations-Infrastruktur für hochdynamische Parallelroboter“ [Koh07] eine Kommunikations-Infrastruktur entwickelt, die eine einheitliche, flexible und modulare Schnittstelle mit zuverlässigen und schnellen Kommunikationsmechanismen für den Aufbau leistungsfähiger PC-basierter Steuerungssysteme für Parallelroboter zur Verfügung stellt. Aufbauend auf dieser

Infrastruktur wurde in [Maa09] eine entsprechende Steuerungsarchitektur für Parallelroboter realisiert.

Diese Arbeit befasst sich mit der Erweiterung und Optimierung der zugrunde liegenden Kommunikations-Infrastruktur hinsichtlich der steuerungstechnischen Ausnutzung des strukturellen Vorteils paralleler Kinematiken.

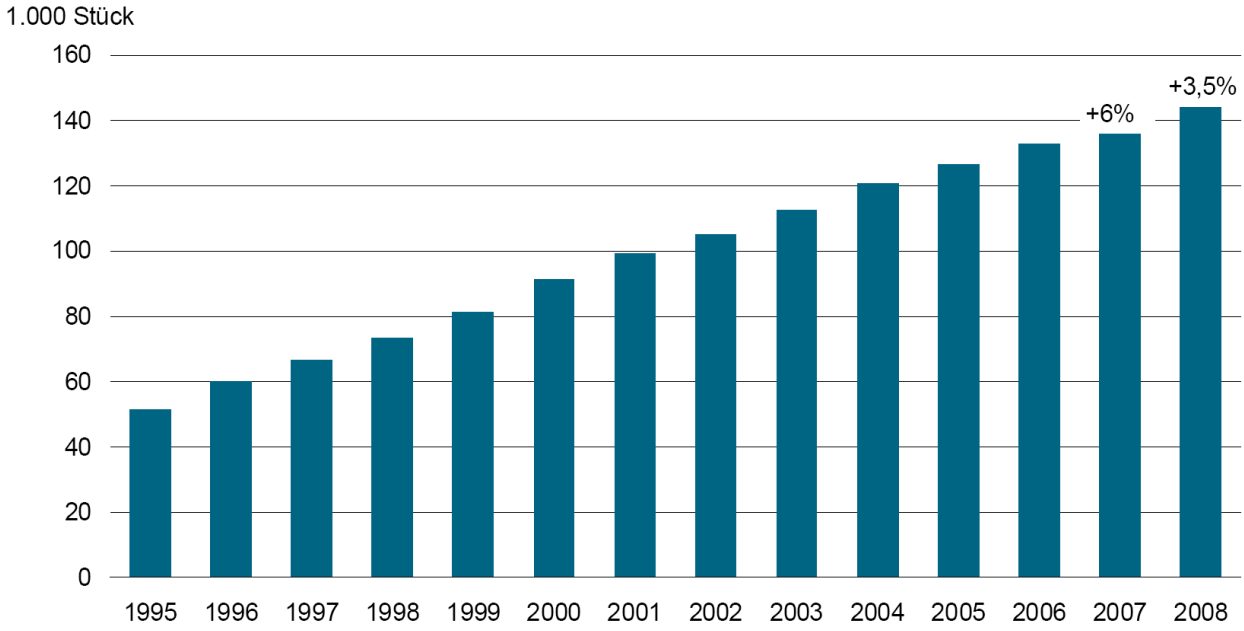

Abbildung 1-1: Geschätzter operierender Bestand Industrieroboter in Deutschland[1]

1.1 Motivation

In den letzten Jahren haben PC-basierte Steuerungssysteme immer mehr Einsatz im Bereich der Robotik gefunden [KUK10] [Mun10] [Gee01] [CLZ+98] [DMM08] [MKH06] [BEC10] [ORO09] [BKM+05]. Neben dem durch den Einsatz von Standard-Komponenten und Schnittstellen verbundenen Kostenvorteil unterliegen Standard-PCs einem ständigen Technologie-Update, was für darauf basierende Robotersysteme eine kontinuierliche Prozessleistungssteigerung und Synchronisationsoptimierung bedeutet. Allerdings stellt die Umsetzung einer PC-basierten Lösung für Steuerungsaufgaben besondere Anforderungen an die Funktionalität und Leistungsfähigkeit der unterlagerten Soft- und Hardwareplattform. Die für die Steuerungssoftware wichtigen Aspekte wie Deterministik, kurze Reaktionszeiten und geringer Jitter müssen bereitgestellt werden.

[1] Quelle: VDMA Robotik

Der Einsatz der PC-Technologie öffnet die Tür zur Entwicklung von neuen offenen und flexibleren Robotersteuerungsansätzen. Beispielhaft sei hier der im Jahre 1981 von Mason [Mas81] und De Schutter [Sv88] eingeführte Task-Frame-Formalismus erwähnt, der einen wichtigen Schritt zur aufgabenorientierten Programmierung von Montageaufgaben [TMH+05] darstellt. Im Gegensatz zur herkömmlichen Bewegungsbeschreibung wird die Bewegung innerhalb eines Task-Frames beschrieben. Das Roboterprogramm wird in atomare Roboterbewegungsbefehle, sogenannte Manipulationsprimitive (Skill Primitive), aufgeteilt.

Aufbauend auf dem Skill-Primitive-Programmierungsansatz lassen sich Steuerungsarchitekturen realisieren, die die Integration von beliebigen Sensoren in den Steuerungsfluss und die Implementierung hybrider Regelungen unterstützen, bei denen jeder Roboterfreiheitsgrad algorithmisch separat geregelt wird. Eine solche Architektur stellt die RCA562[2] [Maa09] [MKH06] [MHS+07] [MSA+08] dar. Sie basiert auf der in der Arbeit von Kohn [Koh07] aufgebauten Kommunikations-Infrastruktur. Die Kommunikations-Infrastruktur besteht aus einer Middleware zur Unterstützung des modularen Aufbaus der Steuerungs-Software und einem auf dem IEEE1394 aufsetzenden Kommunikationsprotokoll zum deterministischen Austausch von Ist- und Sollwerten zwischen dem Steuerungsrechner und den im Roboter eingesetzten Sensoren und Aktoren. Die Kommunikations-Infrastruktur baut auf Standard-Hardwarekomponenten auf; ihre Performanz ermöglicht die Realisierung von Steuerungszyklen mit einer maximalen Frequenz von 1 kHz. Wegen der begrenzten Zyklusfrequenz kann das dynamische Potenzial von Parallelrobotern nicht voll ausgenutzt werden. Denn je schneller der Manipulator bei unverminderter Wiederholgenauigkeit bewegt werden soll, umso größer muss die Zyklusfrequenz ausfallen. Da die Zyklusfrequenz maßgeblich durch Datenübertragung und Berechnungszeiten bestimmt wird, bedarf die bessere Ausnutzung der dynamischen Eigenschaften von Parallelrobotern einer Optimierung der durch die Kommunikations-Infrastruktur bereitgestellten Kommunikationsmechanismen.

In [Koh07] wurde bereits die Notwendigkeit einer Verteilung der bestehenden Robotersteuerung auf mehrere Rechner erwähnt. Das vorrangige Ziel hierbei ist die Realisierung komplexerer und rechenzeitintensiverer Bewegungsalgorithmen sowie die Integration von Kamerasystemen in die Steuerung. Eine derartige Erweiterung der Robotersteuerung erfordert die Verteilung der zugrunde liegenden Kommunikations-Infrastruktur von einem zentralen Rechneransatz hin zu einem auf einem Rechnernetzwerk basierenden Ansatz. Ein vergleichbares System auf dem Forschungsgebiet der Steuerungstechnik für Roboter bietet die auf MiRPA [FKK+07] [FKW10] basierende Steuerungsarchitektur[Fin04]. Leider kann dort das Erreichen des zeitlichen Determinismus im verteilten System unter anderem wegen der zugrunde liegenden ethernetbasierten Bustechnologie ohne Synchronisation nicht garantiert werden.

[2] **R**obot **C**ontrol **A**rchitecture of Collaborative Research center **562**

Das Forschungsgebiet des autonomen Computings [HAH09] [KC03] bietet Methoden und Konzepte, die Systeme in die Lage versetzen, selbständig auf äußere Einflüsse zu reagieren und optimale aufgabenorientierte Änderungen der Systemkonfiguration vorzunehmen. Angesichts des hohen Gewinns hinsichtlich Performanz und Verwaltungsaufwand ist die Integration einer solchen Eigenschaft in der auf einem Rechnernetzwerk basierenden verteilten Steuerungsarchitektur eine sinnvolle Ergänzung.

1.2 Zielsetzung

Die vorliegende Arbeit entstand im Rahmen des Sonderforschungsbereichs (SFB) 562 „Robotersysteme für Handhabung und Montage – Hochdynamische Parallelstrukturen mit adaptronischen Komponenten“. Innerhalb dieses interdisziplinären Forschungsprojekts werden methoden-, komponenten- und systembezogene Grundlagen für die Entwicklung von Parallelrobotern erarbeitet. Die Forschungsergebnisse dieser Arbeit sollen einen Beitrag zur Steigerung der Performanz und Zuverlässigkeit offener PC-basierter Steuerungssysteme insbesondere im Bereich der Parallelrobotik liefern.

In besonderem Maße befasst sich diese Arbeit mit der Optimierung der bereits bestehenden Kommunikations-Infrastruktur [Koh07] [DMM+07] [DMK+10] [MDM08]. Die Optimierung soll die Realisierung kürzerer Steuerungszyklen, und damit eine bessere Ausnutzung des strukturbedingten Potenzials von Parallelrobotern durch höhere Verfahrgeschwindigkeiten und Beschleunigungen, ermöglichen. Die Optimierung muss sich sowohl auf die Soft- als auch auf die Hardware beziehen.

Im Zuge der Software-Optimierung werden die *Leistungsfähigkeit* und die *Funktionssicherheit* der von der Middleware MiRPA-X und dem Kommunikationsprotokoll IAP bereitgestellten Funktionalitäten verbessert. Bezüglich der Middleware werden Entwurfsmuster und Kommunikationsmechanismen so erweitert und geändert, dass die darauf aufbauende modulare Steuerungssoftware ein vereinfachtes Design mit kürzeren Reaktionszeiten aufweisen kann. Außerdem soll durch die Software-Optimierung eine bessere Ausnutzung der von Mehrkernprozessoren bereitgestellten Rechenleistung erreicht werden. Weiter soll die Betriebszuverlässigkeit des Kommunikationsprotokolls erhöht werden. Dies lässt sich durch die Integration fehlertoleranter Mechanismen erreichen, die die Auswirkung gelegentlich auftretender Fehler auf die zyklische Kommunikation unterdrücken und den Steuerungszyklus auch im Fehlerfall aufrechterhalten.

Die Hardware-Optimierung befasst sich mit den Kommunikationsmodulen, die zur Einbindung von Roboter-Sensor- und Aktor-Einheiten eingesetzt werden. Hierbei soll die wegen des Einsatzes von Standard-Komponenten verbesserungsfähige *Datenverarbeitung* durch eine gezielte Integration von Hardware-Komponenten in einer auf FPGA-Technologie basierende dedizierte Plattform beschleunigt werden.

Ein weiteres Ziel dieser Arbeit ist die Erweiterung der Kommunikations-Infrastruktur hin zu einem *skalierbaren verteilten System*, in dem die Rechenleistung eines Rechnernetzwerks anstatt des bisherigen einzigen Rechners eingesetzt wird. Die Middleware soll so erweitert werden, dass sie Anwenderapplikationen eine transparente Nutzung von Kommunikationsmechanismen im verteilten System gewährleistet. Damit verteilte Rechenprozesse in den Steuerungszyklus eingebunden werden können, müssen die Kommunikationsmechanismen eine kurze Latenzzeit und ein Echtzeitverhalten aufweisen. Dies wirft die Frage nach einem geeigneten echtzeitfähigen Bussystem zum Aufbau des Rechnernetzwerks auf. Diese Frage wird ebenfalls im Rahmen dieser Arbeit geklärt.

Um eine optimale Nutzung der im verteilten System verfügbaren Ressourcen zu gewährleisten, soll im Rahmen des SFB562 ein Self-Manager [MHS+07] [MSA+08] [SH07] [SHG07] in die Steuerungssoftware exemplarisch integriert werden. Der Self-Manager soll selbständig und effizient auf Veränderungen in der Systemtopologie sowie auf sich ändernde Anforderungen an die Steuerung reagieren und eine automatische Laufzeitanpassung der Verteilung der Steuerungsmodule ausführen können. In diesem Zusammenhang wird im Rahmen dieser Arbeit eine *Monitoring-Komponente* in die Middleware integriert, die topologische und zeitliche Informationen über die Ausführung von Tasks im verteilten System zur Laufzeit ermittelt und dem Anwender bereitstellt. Sie dient als Grundlage für die Entwicklung des Self-Managers.

1.3 Gliederung der Arbeit

Gegenstand dieser Arbeit ist die strukturelle und performanzorientierte Optimierung und Erweiterung der für die Entwicklung von Steuerungsarchitekturen für Parallelroboter im Rahmen des SFB562 zugrunde liegenden Kommunikations-Infrastruktur.

Im folgenden Kapitel wird zunächst der Aufbau der Kommunikations-Infrastruktur erläutert. Das Design und die Funktionalitäten der einzelnen Komponenten sowie die bereitgestellten Kommunikationsmechanismen werden zum besseren Verständnis der in den weiteren Kapiteln umgesetzten Optimierungs- und Erweiterungsmaßnahmen erläutet. Anschließend wird die im Rahmen des SFB562 entwickelte Steuerungsarchitektur RCA562 mit der erreichbaren Performanz kurz dargestellt.

In Kapitel 3 werden die einzelnen Funktionalitäten in Soft- und Hardware innerhalb der Kommunikations-Infrastruktur auf deren Optimierungspotenzial untersucht. Die Untersuchung umfasst sowohl das auf dem Steuerungsrechner zugrunde liegende Software-Framework (bestehend aus MiRPA-X, IAP und IEEE1394-Software-Treiber) als auch die Hard- und Software-Komponente der Kommunikationsmodule. Anschließend werden für jede untersuchte Komponente Optimierungsmaßnahmen formuliert, die einen Beitrag zur Steigerung der gesamten Zyklusfrequenz der Steuerung leisten können.

Kapitel 4 behandelt die technische Umsetzung der im Kapitel 3 zu dem Software-Framework formulierten Optimierungsmaßnahmen.

Kapitel 5 befasst sich mit der technischen Umsetzung der im Kapitel 3 zu den Kommunikationsknoten formulierten Optimierungsmaßnahmen. Die Maßnahmen umfassen die Realisierung eines neuen Hard- und Software-Designs. Anschließend werden die Auswirkungen der Optimierungsmaßnahmen auf die gesamte Systemperformanz dargestellt.

In Kapitel 6 erfolgt die Beschreibung der Entwicklung und Implementierung der verteilten Version von MiRPA-X. Die Besonderheiten bei diesem verteilten System sind der Einsatz des IEEE1394 Standards als zugrunde liegende Bustechnologie zur Kopplung der verteilten Rechner und die dadurch erzielbare Performanz und Echtzeitfähigkeit für verteilte Anwendungen.

In Kapitel 7 werden schließlich im Rahmen des System-Monitorings die Mechanismen beschrieben, die im Zuge der Integration von autonomen Eigenschaften in das verteilte Steuerungssystem in die Middleware integriert wurden. Der Systemmonitor wurde so entworfen, dass die Systemperformanz durch die Ermittlung der topologischen und zeitlichen Informationen nicht oder im schlimmsten Fall nur geringfügig beeinträchtigt wird. Anschließend wird anhand eines Applikationsbeispiels der Beitrag des Systemmonitors zu dem Aufbau einer im Rahmen des SFB562 entwickelten neuartigen und selbstoptimierenden Steuerungssoftware beschrieben, die über im Netzwerk verteilte Softwarekomponenten verfügt.

Schließlich werden die einzelnen Ergebnisse in Kapitel 8 zusammengefasst und weiterführende Arbeiten im Rahmen eines Ausblicks genannt, die aufbauend auf die erzielten Ergebnisse umgesetzt werden können.

2 Ausgangszustand der Kommunikations-Infrastruktur

In diesem Kapitel erfolgt die Beschreibung der ursprünglichen Kommunikations-Infrastruktur, die im Rahmen des SFB562 für die Steuerung von Parallelrobotern entwickelt wurde. Obwohl die Arbeiten von Beckmann [Bec01] und Kohn [Koh07] die im Bereich der Robotik eingesetzten Kommunikations-Infrastrukturen beinhalten, wird hier erneut eine kurze zusammenfassende Darstellung zunächst gegeben, weil dies das Verständnis für Anforderungen an die Optimierungsaufgabe erleichtert.

2.1 Übersicht vorhandener Infrastrukturen

In der Literatur findet man viele Infrastrukturen, die generell für die Entwicklung von Robotersteuerungen zum Einsatz kommen. Im Folgenden werden nur diejenigen betrachtet und für den Einsatz in einer Robotersteuerung für Parallelkinematiken bewertet, die über eine offene Programmierschnittstelle verfügen. Neben der verfügbaren Performanz wird die Eignung der angebotenen Software-Mechanismen für die Realisierung komplexer Robotersteuerungen für Parallelkinematiken als Bewertungskriterium festgelegt.

CORBA/TAO (ACE)

CORBA (Common Object Request Broker Architecture) ist die Spezifikation einer objektorientierten Middleware [Fis02, Cor12], die verteilte Applikationen ermöglicht, unabhängig von verwendeter Programmiersprache, Hard- und Softwareplattformen und Netzwerkverbindungen, miteinander zu kommunizieren. TAO [Cor12] ist eine frei verfügbare Implementierung des Realtime-CORBA Standards. Die Implementierung von TAO erfolgt aud der Basis von Komponenten des Adaptiven Computing Environment (ACE) [Cor12]; eine freie plattformunabhängige Programmbibliothek, die verschiedene Netzwerkprogrammierungsmodule kapselt. TAO stellt deterministische Kommunikationsmechanismen zur Verfügung und verbessert den CORBA-Ereignisdienst um wichtige Eigenschaften, wie die Echtzeit-Ereignisverarbeitung und Ablaufsteuerung. Es ist aber für Applikationen ausgelegt, die große Datenmengen miteinander austauschen. Für die Übertragung von einem 128 Kbytes großen Datenpuffer sind beispielweise über 90 ms notwendig [Cor12]. Weiterhin ist die Datenübertragungszeit wegen mangelnder deterministischen Mechanismen der ethernet-basierenden physikalischen Schicht nicht garantiert. Für die Realisierung von Steuerungszykluszeiten kleiner 1ms ist die CORBA/TAO-Lösung aus diesem Grund nicht geeignet.

OSACA

OSACA (Open System Architecture for Controls within Automation Systems) [PW97] bezeichnet eine offene Referenzarchitektur, die für Bewegungssteuerungen unter Windows NT und VxWorks eingesetzt werden kann. Für die Realisierung einer Robotersteuerung für

Parallelkinematiken ist der zu hohe Abstraktionsgrad der bereitgestellten Mechanismen ungeeignet. Die Realisierung der nötigen Zyklusfrequenz ist aufgrund nicht existierender Performanzangaben nicht garantiert.

OROCOS

OROCOS (Open Robot Control Software) [ORO09] ist eine Open Source Softwareplattform, welche die Entwicklung generischer und modularer Echtzeit-Softwarekomponenten für Roboter- und Maschinensteuerungen ermöglicht. Sie stellt folgende Funktionalitäten für die Komponentenentwicklung zur Verfügung:

- Commands: asynchroner Mechanismus zur Ausführung räumlich verteilter Tasks.
- Methods: zum entfernten Objektzugriff.
- Properties: xml-basierende modifizierbare Laufzeitparameter einer Komponente.
- Events: zur Ausführung ereignisbasierter Funktionen.
- Data Flow Port: als Transportmechanismus für gepufferte oder ungepufferte Daten.

Der asynchrone Datenaustausch zwischen zwei Kommunikationspartnern dauert zwischen 100 und 1000 µs [SOE06]. Ferner erlauben die zu statisch ausgelegten Softwarekonstrukte nicht die Umsetzung komplexeren Steuerungskonzepten, wie sie bei der Steuerung hoch dynamischer Parallelkinematiken notwendig sind.

MIRO

MIRO (Middleware for Mobile Robotic Application) [USE+02] ist eine Middleware, die auf mobile Robotik optimierte Funktionalitäten bereitstellt. MIRO unterstützt sowohl die Publisher/Subscriber- als auch die Client/Server-Kommunikation, allerdings weist es eine hohe Kommunikationslatenz (mehr als 12 ms für die Übertragung einfacher Bewegungsbefehle [USE+02]) auf.

CLARITy

CLARITy (Layer Architecture for Robotic Autonomy) [NAS10] ist ein Framework, das die Wiederverwendung von Roboter-Software und die Interoperabilität von Komponenten und Algorithmen in heterogenen Roboterplattformen unterstützt. CLARITy wird hauptsächlich in der mobilen Robotik eingesetzt und nutzt sowohl TAO als auch TCP als Kommunikationsschnittstelle. Leistungsmerkmale von CLARITy sind in der Literatur nicht vorhanden. Aufgrund des in der Wiederverwendbarkeit von Robotersoftware festgelegten Schwerpunkts und der eingesetzten Kommunikationsschnittstelle ist die für die Steuerung von

Parallelkinematiken benötigte Flexibilität sowie die Performanz der bereitgestellten Kommunikationsmechanismen nicht garantiert.

ORCA2

ORCA2 (Organic Robotic Control Architecture) [BKM+05] [Orc08] ist ein Open Source Framework für die Entwicklung von komponentenbasierten Robotersteuerungssystemen. ORCA2 stellt eine Reihe von Komponenten (von Steuerungsalgorithmen bis Treibersoftware) zur Verfügung, durch deren geschickte Zusammensetzung sich komplexe Robotersteuerungssysteme aufbauen lassen. Alle Komponenten kommunizieren über ICE (Internet Communication Engine) [Zer10]. Wegen seiner Leistungsmerkmale (z.B. über 70µs für lokale Client/Server-Kommunikation mit 10 Bytes Nutzdaten) ist ORCA leider für die Steuerung von Parallelrobotern nicht geeignet.

PLAYER

PLAYER [pla10] ist ein plattformunabhängiger Netzwerk-Server, der der Roboterapplikation eine einfache TCP/IP-basierende Schnittstelle für die Integration einer großen Anzahl von Sensoren und Aktoren bereitstellt. Mit Hilfe des Client/Server-Modells lassen sich flexible und modulare Robotersteuerungen aufbauen. User-Applikationen können als Client über ein TCP-Socket den Player-Server ansprechen, um Sensordaten auszulesen, Aktoren Sollwerte zu übermitteln oder die vorhandenen Module zu konfigurieren. Obwohl PLAYER einen hohen Grad an Flexibilität für die Entwicklung von Steuerungsapplikationen aufweist, kommt es wegen der mit TCP/IP verbundenen Kommunikationslatenzen für die Robotersteuerung von Parallelkinematiken nicht in Frage.

Fazit

Zusammenfassend lässt sich feststellen, dass die vorhandenen Infrastrukturen nicht den Ansprüchen von Robotersteuerungen für Parallelkinematiken genügen. Dies liegt sowohl an den statischen Konstrukten, welche die Flexibilität der Robotersteuerung begrenzen, als auch an der unzureichenden Performanz der bereitgestellten Kommunikationsmechanismen, die sich als Nachteil für die Realisierung von für Parallelkinematiken notwendigen kurzen Steuerungszyklen darstellen.

2.2 Übersicht über die SFB562-Kommunikations-Infrastruktur

Innerhalb des SFB562 kommt für die Realisierung der Steuerung von Parallelrobotern ein zentraler PC-basierter Steuerungsansatz zum Einsatz. Sämtliche für die Steuerung des Roboters notwendige Berechnungen werden in modularen Softwarekomponenten gekapselt und erfordern

enge Wechselwirkungen je nach Komplexität der Steuerungsaufgabe, Roboterstruktur und Roboterdynamik. Im Rahmen des SFB562 wurde eine umfassende Kommunikations-Infrastruktur definiert und entwickelt, die neben der Unterstützung von Modularität und Flexibilität im Software-Entwicklungsprozess einen deterministischen Prozessablauf (Scheduling) und die notwendige Echtzeit-Kommunikation zwischen lokalen und externen Softwarekomponenten ermöglicht.

Mit dieser Kommunikations-Infrastruktur lassen sich im Allgemeinen Regelungszyklen mit theoretisch maximalen Frequenzen bis zu 8 kHz unter Echtzeitbedingungen realisieren. Abbildung 2-1 stellt die Kommunikationsarchitektur dar. Sie basiert auf einem modularen Ansatz mit den folgenden Komponenten:

- Kommunikations-Middleware MIRPA-X (Middleware for Robotic and Process Control Applications - extended)
- Industrial Automation Protocol (IAP) als deterministisches Kommunikationsprotokoll zum Echtzeitdatenaustausch zwischen dem Steuerungsrechner und den externen Komponenten
- IEEE1394 (FireWire) Standard als echtzeitfähiger Kommunikationsbus mit hohem Datendurchsatz

Um den Entwurf und die Implementierung von modularer Steuerungssoftware zu unterstützen, wird die Kommunikations-Middleware MiRPA-X als Kernfunktionalität eingesetzt. Für die Inter-Modul-Kommunikation übernimmt MiRPA-X sämtliche kommunikationstechnische Implementierungsdetails und ermöglicht es dem Anwender, sich ausschließlich auf den algorithmischen Teil der Modulsoftware zu konzentrieren. In der MiRPA-X-Umgebung werden Steuerungssoftwaremodule entweder als reine Client- oder Server-Prozesse eingesetzt, die die Steuerungsfunktionalität über Synchronisationsdienste, synchrone und asynchrone nachrichten-basierte Kommunikations- sowie Shared-Memory-Dienste kooperierend realisieren.

Die Steuerungssoftware ist über den IEEE1394-Bus mit den externen DSP-basierenden Kommunikationsknoten (KK) verbunden, die als Zwischenglied zwischen dem Steuerungsrechner und den Sensor- und Aktoreinheiten des Roboters dienen. Um einen festen Steuerungszyklus zu realisieren und einen deterministischen Datenaustausch zwischen der Steuerung und den KK zu gewährleisten, wird das speziell auf den IEEE1394 Standard zugeschnittene IAP-Protokoll eingesetzt. Instanzen des IAP-Protokolls werden sowohl auf dem Steuerungsrechner als auch auf den KKs umgesetzt. Innerhalb eines Steuerungszyklus greift das IAP über von MiRPA-X bereitgestellte Dienste auf von der Steuerung generierte Sollwerte zu und sendet sie zu den KKs, deren IAP-Instanzen im Gegenzug Istwerte sammeln und an den Steuerungsrechner zurücksenden. Die Transaktionen zwischen dem zentralen Steuerungsrechner und den KKs realisiert das IAP über die echtzeitfähigen asynchronen Mechanismen des IEEE-1394-Standards.

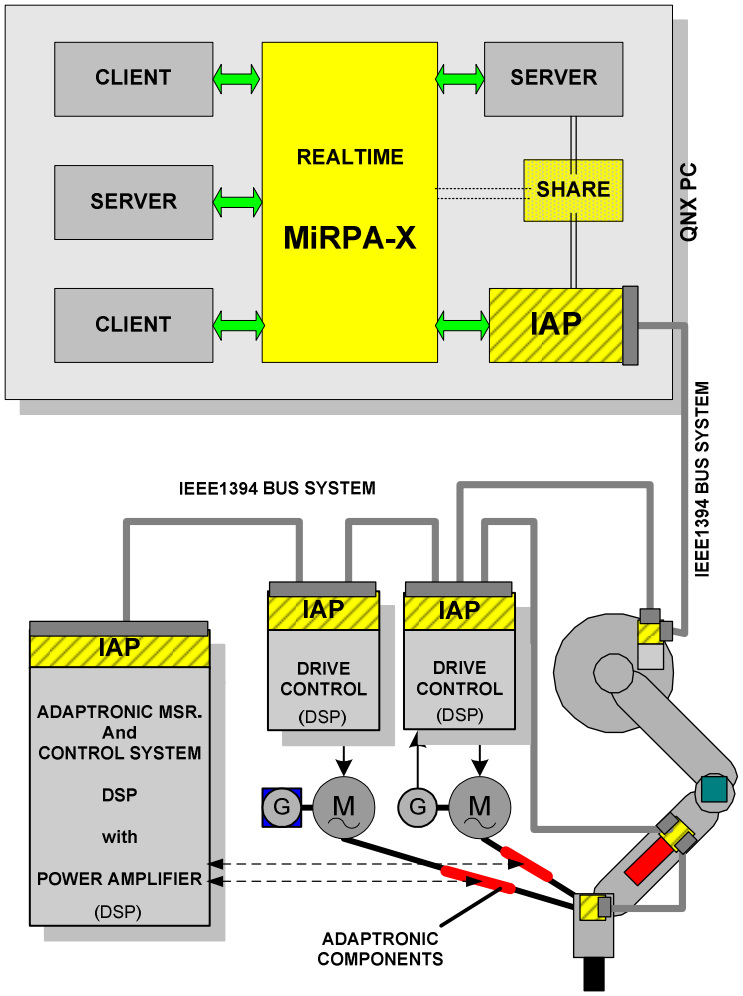

Abbildung 2-1: Software- und Kommunikationsarchitektur

2.3 Middleware MIRPA-X

Die Motivation zur Entwicklung einer dedizierten Kommunikations-Middleware zur Unterstützung der Entwicklung modularer Steuerungssoftware im Rahmen des SFB562 wurde in [Koh07] ausgeführt. Dort wurden unter anderen die unzureichende Leistungsfähigkeit und die nicht zufriedenstellenden Synchronisationsmechanismen existierender Systeme als Grund genannt. Entwicklungsergebnisse von MiRPA-X wurden ebenfalls vorgestellt. In diesem Abschnitt werden für ein besseres Verständnis der in den nachfolgenden Kapiteln eingeführten Optimierungen und architektonischen Erweiterungen lediglich grundlegende Funktionen von MiRPA-X beschrieben. Für eine detaillierte Beschreibung sei auf [Koh07] verwiesen.

MiRPA-X ist ein Kernelement der im SFB562 angewandten Kommunikationsarchitektur. Sie ist speziell für den Einsatz in der Entwicklung von Steuerungssoftware für Parallelroboter zugeschnitten, daher ist die Implementierung bereitgestellter Funktionalitäten auf Performanz optimiert. MiRPA-X unterstützt einen modularen Softwareentwurf und ermöglicht einen vollständig transparenten Datenverkehr zwischen Softwaremodulen. Ferner unterstützt sie die

Umsetzung von Publisher/Subscriber- sowie Client/Server-Kommunikationsmodellen. Mit dem konsequent umgesetzten Client/Server-Kommunikationsmodell, bei dem jeder Anwenderprozess entweder als Client oder als Server registriert werden muss, wird bereits auf der Entwurfsebene für die Deadlock-Freiheit der entwickelten Applikationen gesorgt. Zur Unterstützung der Kommunikation zwischen den Steuerungssoftwaremodulen stellt MiRPA-X sowohl nachrichtenbasierte als auch shared-memory-basierte Kommunikationsmechanismen zur Verfügung. Außerdem stellt MiRPA-X weitere Mechanismen bereit, mit deren Hilfe sich Softwaremodule synchronisieren lassen.

Der strukturelle Aufbau von MiRPA-X besteht aus zwei Hauptkomponenten, die als unabhängige Threads implementiert sind: der ObjectServer und der Scheduler. Der ObjectServer wickelt alle nachrichtenbasierten Kommunikationsvorgänge ab und verwaltet die Shared-Memory-Zugriffe sowie die Synchronisierungsvariablen der Applikationsprozesse. Er wertet Client-Anfragen aus und leitet sie bei Bedarf an entsprechende Serverprozesse weiter. Die Weiterleitung der Anfragen erfolgt anhand einer zwischen Client und Server abgestimmten Vereinbarung, die für die jeweiligen Applikationsdienste einen systemweit eindeutigen Namen festlegt. Durch den Einsatz dieses Namensdienstes ist es möglich, modulare Applikationsdienste zu implementieren, ohne die Identität und den aktuellen Standort des zugehörigen Zielservers kennen zu müssen. Der Scheduler seinerseits sorgt dafür, dass gesonderte Softwaremodule in einer einstellbaren Reihenfolge sequentiell, zyklisch und in Echtzeit mit einer höheren Priorität ausgeführt werden.

2.3.1 Nachrichtenbasierte Kommunikation

Der nachrichtenbasierte Kommunikationsmechanismus von MiRPA-X basiert auf dem von QNX bereitgestellten Message-Passing. Unterschieden wird zwischen synchronen REQUEST- und asynchronen COMMAND-Nachrichten. Mit dem umgesetzten Client/Server-Modell dürfen beide Nachrichtentypen ausschließlich von Client-Applikationen gesendet und von Server-Applikationen empfangen werden. Dementsprechend darf ein Server keine Nachricht an einen anderen Server senden.

Der Nachrichtentyp REQUEST wird deshalb als synchron bezeichnet, weil jeder aufrufende Client so lange blockiert, bis der entsprechende Server die Anfrage bearbeitet und beantwortet hat. Erst danach kann der Client weiter ausgeführt werden und die Antwort des Servers auswerten. Eine weitere für den Anwender entscheidende Eigenschaft der REQUEST-Nachricht ist, dass Applikationsdaten nicht mit der Anfrage an den Server übertragen werden können. Falls dies aber notwendig ist, müssen die Daten auf einem anderen Weg (Shared-Memory oder gesonderte COMMAND-Nachricht) dem Server übermittelt werden.

Der asynchrone Nachrichtentyp COMMAND ist für einen aufrufenden Client nicht blockierend. Das heißt, dass der Client nach dem Versand der Nachricht weiter ausgeführt werden kann, während der zugehörige Server die empfangene Nachricht bearbeitet. Hierbei muss angemerkt

werden, dass mehr als ein Server auf den Empfang derselben Nachricht blockieren kann, was bei REQUEST-Nachrichten nicht der Fall ist. Auf eine Nachricht mit dem Typ COMMAND kann nicht geantwortet werden. In der Praxis lassen sie sich dafür einsetzen, Applikationsdaten von einem Client zu einem oder mehreren Servern zu übermitteln, oder mehrere Server gleichzeitig zu aktivieren.

2.3.2 Shared-Memory basierte Kommunikation

Neben dem nachrichtenbasierten Kommunikationsmechanismus bietet MiRPA-X die Anwendung von Shared-Memory-Bereichen als weiteren Inter-Prozess-Kommunikationsmechanismus. Client- und Serverprozesse kommunizieren miteinander, indem sie auf über Prozessgrenzen hinweg nutzbare Speicherbereiche zugreifen. Dieser Zugriff vollzieht sich augenblicklich (unmittelbar und ohne Blockierung) und wird daher vornehmlich bei besonders schnell abzuwickelnden Vorgängen eingesetzt. Auch in diesem Fall erfolgt die Identifizierung über den Einsatz des Namendienstes: jeder Shared-Memory-Bereich lässt sich über einen eindeutigen Namen ansprechen.

MiRPA-X stellt die Shared-Memory-Bereiche den kooperierenden Prozessen zu Verfügung, beteiligt sich aber nicht weiter an der eigentlichen Kommunikation. Falls mehrere Prozesse konkurrierend auf denselben Shared-Memory-Bereich zugreifen wollen, muss dessen Datenintegrität sichergestellt werden. Für einen sicheren Datenzugriff stellt MiRPA-X geeignete Synchronisationsmechanismen (Condition Variable, Mutex, …) und eine Prozessablaufsteuerung (vom Scheduler-Thread umgesetzt) bereit.

2.3.3 Synchronisationsmechanismen

Alle von MiRPA-X bereitgestellten Synchronisationsmechanismen basieren auf entsprechenden QNX-Funktionalitäten. Sie werden alle über einen einheitlichen Namensdienst referenziert, so dass die Synchronisationsobjekte im gesamten Steuerungssystem bekannt sind und sichergestellt werden kann, dass nicht versehentlich auf unbeabsichtigte Daten zugegriffen wird. Es werden folgende Synchronisationsmechanismen unterschieden:

- **Mutex** – um sicherzustellen, dass auf kritische Bereiche (z.B. Shared-Memory-Bereich zwischen mehreren Prozessen) in strikt einzelner und prioritätsbezogener Weise zugegriffen wird, können solche Zugriffe auf Applikationsebene per Mutex verriegelt werden. Die Synchronisierung funktioniert nur dann, wenn die Vereinbarung zur Nutzung der Verriegelung von allen beteiligten Prozessen eingehalten wird. Wenn ein Prozess A das Mutex verriegelt hat, muss jeder weitere Prozess blockiert warten, bis der Prozess A das Mutex freigibt, bevor er ebenso auf den Speicherbereich zugreifen kann.

- **Condition Variable** – durch den Einsatz dieses Mechanismus können Prozesse solange blockiert werden, bis spezielle, vorher vereinbarte und über Prozessgrenzen hinweg geltende Applikationszustände eingetreten sind.
- **Event Collection** – durch den Einsatz dieses auf Semaphoren basierenden Mechanismus kann die Ausführung einer Applikationsfunktionalität solange verzögert werden, bis eine vorher festgelegte Anzahl spezifischer Ereignisse eingetreten ist.
- **Puls** – mit diesem Mechanismus lassen sich auf Nachrichtenempfang blockierte Prozesse mit einer frei wählbaren Priorität wieder aktivieren.

2.3.4 Prozessablaufsteuerung (MiRPA-X-Scheduler)

MiRPA-X unterstützt die sequenzielle und zyklische Ausführung dedizierter Applikationsmodule. Dazu dient eine Token-basierte Prozessablaufsteuerung (MiRPA-X-Scheduler), die sich aber deutlich von dem Standard-QNX-Scheduler des unterlagerten Betriebssystems unterscheidet. Sie verknüpft und vereinheitlicht die unterlagerten QNX-Funktionalitäten und sorgt dafür, dass Applikationsmodule auf einer hohen Prioritätsebene in applikationsspezifisch konfigurierter Reihenfolge sequentiell und in garantierter Echtzeit abgearbeitet werden. In diesem Zusammenhang werden die zyklisch ausgeführten Applikationsmodule Token-Threads genannt. Um die sequenzielle Ausführung der Token-Threads zu realisieren und das Erfüllen ihrer Echtzeitbedingungen zu ermöglichen, fordert der Scheduler über eine prioritätsbasierte Steuerung die nötige Bandbreite (CPU-Zeit) aus der eingestellten Steuerungszykluszeit ein. Durch die höhere Priorisierung der Scheduler-Aktivitäten lässt sich eine deterministische Ablaufsteuerung realisieren, die von den auf niedriger Priorität laufenden nachrichtenbasierten Applikationen nicht unterbrochen werden kann. Weiterhin lässt sich die Reihenfolge der Ausführung von Token-Threads durch den Scheduler dynamisch vom Anwender modifizieren.

MiRPA-X verwendet zur Realisierung des Schedulers einen nachrichtenbasierten Tokenpassing-Mechanismus. Token-Threads melden ihre Bereitschaft zur Aktivierung durch einen entsprechenden MiRPA-X-Funktionsaufruf an. Die Reihenfolge der Ausführung im Token-Zyklus ist unabhängig von der Reihenfolge der erfolgten Anmeldung und bezieht sich auf eine feste Vereinbarung, die zu Anfang eines jeden Token-Zyklus besteht. Der MiRPA-X-Scheduler lässt sich frei mit einem beliebigen externen oder internen Taktgeber verknüpfen, aus dem er die Startimpulse für den Zyklusaufbau ableitet.

Abbildung 2-2 zeigt einen exemplarischen Applikationszyklus, in dem zunächst Token-Threads, die unter harten Echtzeitbedingungen arbeiten, nach Eingang des durch FireWire generierten Startimpulses (CST) zum Ablauf kommen. Das sind Prozesse, die in einem Steuerungssystem immer nach festen Zeitabständen aufgerufen werden müssen, um beispielsweise maschinenabhängige Regelungsfunktionalitäten zu realisieren. Diese Prozesse sind während ihrer Ausführung nicht unterbrechbar. Der Scheduler leitet den Token der Reihe nach an alle Token-Threads weiter, die aufgrund des sequentiellen Schedulings Applikationsdaten

untereinander ohne Verletzung der Datenintegrität austauschen können. Nachdem alle Token-Threads im jeweiligen Zyklus ausgeführt wurden, steht die verbleibende Zykluszeit den nachrichtenbasierten Applikationsprozessen auf niedrigerer Priorität zur Verfügung. Beim nächsten Zyklusstartimpuls fängt die sequentielle Abarbeitung der Token-Threads erneut an.

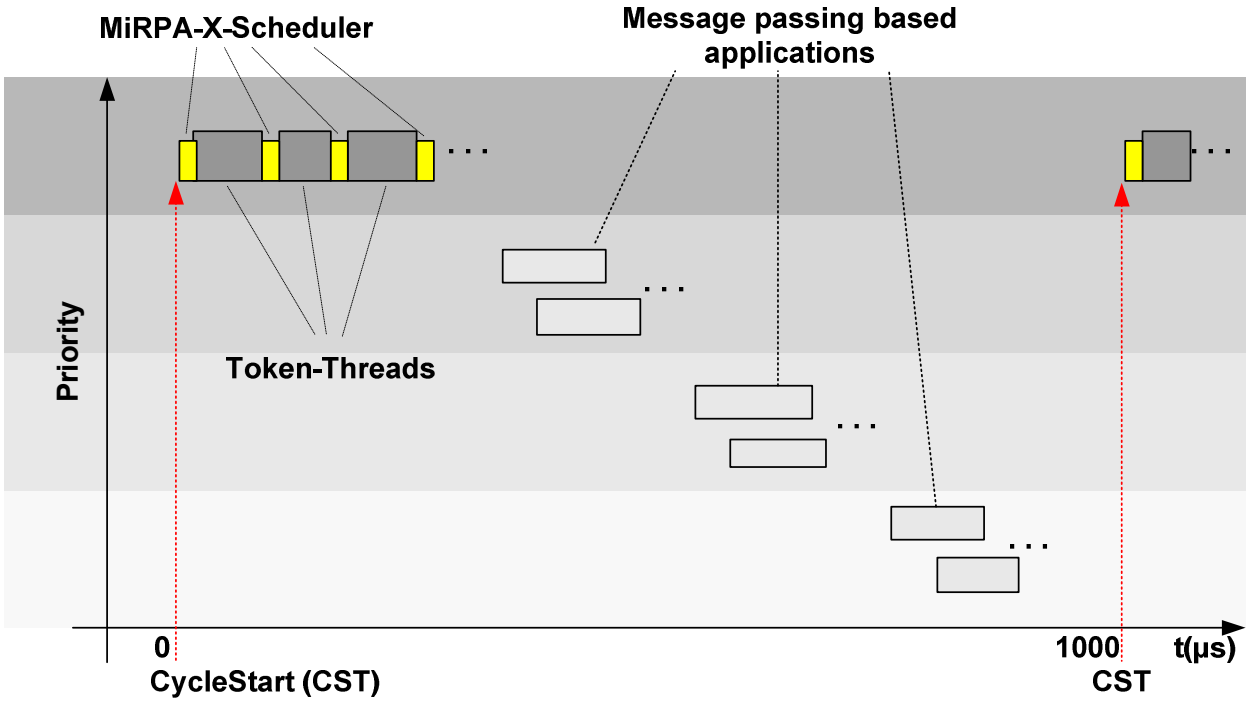

Abbildung 2-2: Exemplarischer Applikationszyklus mit dem MiRPA-X-Scheduler

2.4 Industrial Automation Protocol (IAP)

Das Industrial Automation Protocol (IAP) wurde entwickelt, um ein auf dem seriellen Bus IEEE1394 [IEE08] aufsetzendes Kommunikations-Protokoll für die Datenübertragung zwischen dem Steuerungsrechner und den im Feld verteilten Sensor- und Aktor-Einheiten zu realisieren. Das IAP ermöglicht eine zyklische Kommunikation mit einer Frequenz bis zu 8 kHz und sorgt für die Einhaltung der Echtzeit- und Synchronisationsbedingungen der zyklischen Steuerungsapplikation. Darüber hinaus bietet es folgenden Funktionalitäten:

- Festlegung von Teilnehmerklassen und -Funktionalitäten
- Ressourcen-Management (Bandbreite und Kommunikationskanäle)
- Netzwerk-Management (Zustandsüberwachung und Zustandsänderungen)
- Zugriff auf globale Parameter (Lesen und Schreiben in verteilten Parametersätzen)

- Systemweite Synchronisation (via CST[3])
- Selbst-Konfiguration (automatische Konfiguration aller Teilnehmer)
- Monitoring und Fehlerbehandlung (Funktionsüberwachung von Applikationsmaster und Slave, Umgang mit fehlenden oder späten Telegrammen)

2.4.1 IAP-Teilnehmer

Im IAP gibt es zwei verschiedene Klassen von Teilnehmern mit jeweils unterschiedlichen Funktionalitäten: die IAP-Slaves und der IAP-Master. Die IAP-Slaves, die auf DSP-basierende KK ausgeführt werden, dienen der Anbindung der Sensor- und Aktor-Einheiten an die zentrale Steuerung. Für jeden Roboterantrieb ist ein IAP-Slave vorgesehen. Zusätzlich sind zwei IAP-Slaves für die Verarbeitung von Analog- und Digitalsignalen zuständig. Nach dem Systemstart werden die IAP-Slaves mit aktuellen vom IAP-Master generierten Systemkonfigurationsdaten initialisiert.

Der IAP-Master verwaltet die verfügbare Bandbreite und Kommunikationskanäle auf dem IEEE1394 Bus, die er den vorhandenen IAP-Slaves je nach Bedarf während der Systemkonfiguration zuordnet. Während des aktiven Betriebs sorgt der IAP-Master für die Synchronisation der verteilten Systemkomponenten mittels einer Triggerung des CSTs alle 125 µs. Der IAP-Master kommuniziert mit der Steuerungsapplikation über Shared-Memory-Bereiche und den in Abschnitt 2.3.4 vorgestellten Token-Passing-Mechanismus. Er versorgt die IAP-Slaves zyklisch mit MDT[4], die aus in der Steuerung generierten Sollwerten bestehen. Weiter werden die über DDT[5] übermittelten Istwerte der Slaves eingelesen und ausgewertet.

2.4.2 IAP-Kommunikationszyklus

Abbildung 2-3 zeigt einen IAP-Kommunikationszyklus mit asynchronen MDT und DDTs. Abgebildet werden die Aktivitäten sowohl auf dem Steuerungsrechner als auch auf dem IEEE1394 Bus und exemplarisch auf zwei KKs (im Bild als „Hardware Communication moduls“ gekennzeichnet). Der IAP-Zyklus beginnt mit dem Eingang des CST-Interrupts im Steuerungsrechner. Daraufhin wird der IAP-Master, der gerade im Besitz des Token ist, aktiviert. Er sendet anschließend ein MDT, das die Sollwerte für alle IAP-Slaves in einem zusammenhängenden Datenstrom enthält. Anschließend werden auf dem Steuerungsrechner nach dem in Abschnitt 2.3.4 vorgestellten Ablauf nachrichtenbasierte Applikationsprozesse ausgeführt. Nach Erhalt des MDT im KK werden die jeweiligen Sollwerte extrahiert und an die Aktoren weitergeleitet. Danach werden die Istwerte in DDTs zusammengestellt und an den

[3] Cycle Start Telegram

[4] Master Data Telegram

[5] Device Data Telegram

Master gesendet. Sobald sämtliche erwartete DDT empfangen und deren Inhalt in entsprechende Shared-Memory-Bereiche kopiert wurden, gibt der IAP-Master den Token ab, so dass alle weitere Token-Threads im System zur Ausführung kommen und auf die Istwerte zugreifen können. Nach der Ausführung sämtlicher Token-Threads steht die bis zum nächsten CST verbliebene Zeit wiederum nachrichtenbasierten Applikationsprozessen zur Verfügung.

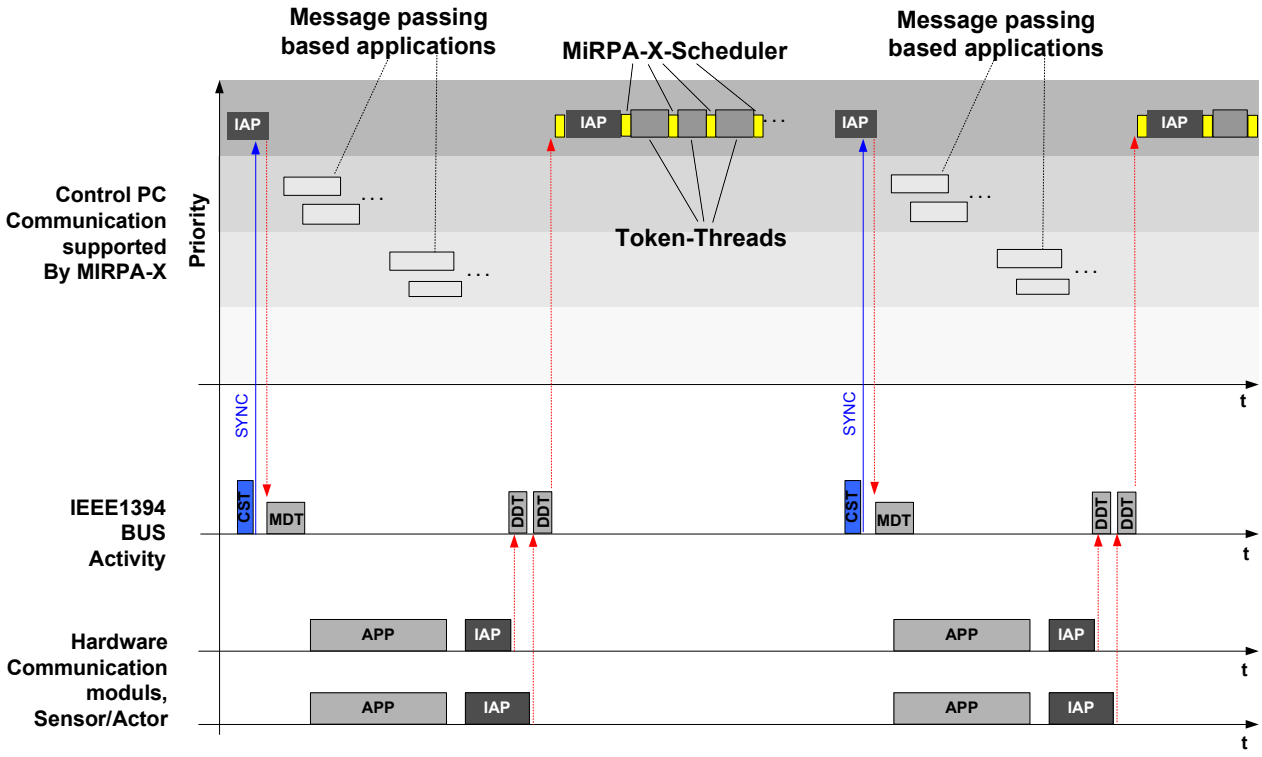

Abbildung 2-3: IAP-Kommunikationszyklus

2.4.3 Synchronisation und Fehlerbehandlung

Das IAP-Protokoll nutzt das vom IEEE1394 Standard generierte jitterarme (weniger als 10 Nanosekunden) CST als Synchronisationssignal. Die infolge des CST-Interrupts durchgeführte Zusammenstellung und Übertragung des MDT erfolgt bei höchster Systempriorität und weist deshalb ebenfalls einen geringen Jitter auf. Aus diesem Grund wird das MDT seitens der IAP-Slaves zusätzlich als Triggersignal für die Ausführung lokaler Anwendungen eingesetzt. Im IAP-Master ist ein Fehlererkennungsmechanismus integriert, der detektieren kann, ob ein IAP-Slave nicht ordnungsgemäß funktioniert und dementsprechend keine oder verzögert DDTs sendet. Dafür prüft der IAP-Master am Ende jedes Kommunikationszyklus, ob sämtliche IAP-Slaves ein DDT gesendet haben. Sollte dies nicht der Fall sein, dann wechselt der IAP-Master in einen Fehlerzustand und bricht sofort die zyklische Kommunikation mit den IAP-Slaves ab.

2.4.4 Automatische Konfiguration

Als Protokollschicht zwischen der Steuerung und dem IEEE1394-Softwaretreiber übernimmt das IAP die Konfiguration des Kommunikationssystems. Sie baut die Kommunikation zwischen dem auf dem Steuerungsrechner laufenden IAP-Master und den auf den KKs ausgeführten IAP-Slaves auf. Der Aufbau erfolgt nach dem in Abbildung 2-4 dargestellten Zustandsdiagramm soweit automatisch, dass Anwender-Interaktionen während des gesamten Vorgangs nicht notwendig sind.

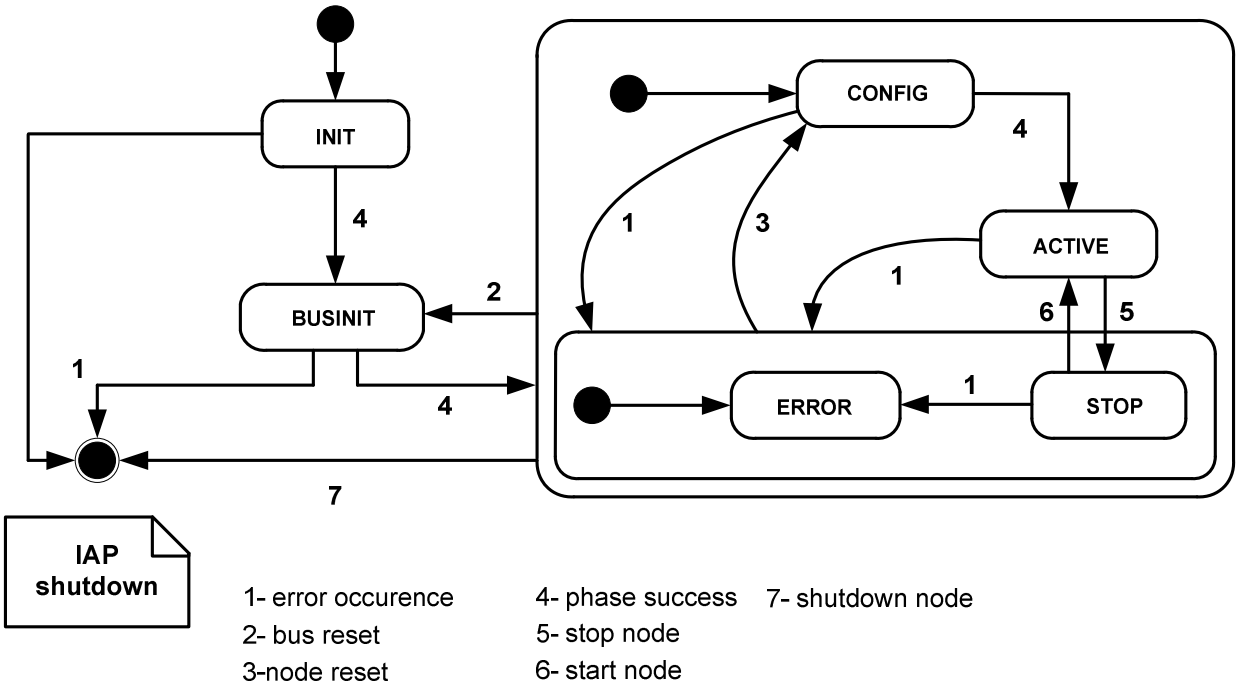

Abbildung 2-4: Zustandsdiagramm eines IAP-Teilnehmers

Nach dem Start werden im Zustand INIT grundsätzliche Software-Initialisierungen ausgeführt. Ebenso werden Speicherbereiche zum Versenden und Empfang von Telegrammen angelegt. In den Zustand BUSINIT wird jedes Mal nach dem Eintritt eines Bus-Reset gewechselt. Dort finden die mit dem IEEE1394 Standard verbundenen Initialisierungen statt, Bus- und Teilnehmer-ID werden ermittelt. Fehler, die in dem Zustand INIT oder BUSINIT auftreten, führen unmittelbar zu einem Abbruch der IAP-Initialisierung.

Nach einem erfolgreichen Abschluss der Businitialisierung wird in den Zustand CONFIG gewechselt. In diesem Zustand wartet der IAP-Master darauf, dass sämtliche IAP-Slaves ihren internen Status mittels IAP-Statustelegramm übermitteln. Danach sendet er user-spezifische Konfigurationsdaten (inkl. Zyklusfrequenz, Kommunikationsmodus, Größe der MDT- und DDT-Telegramme) und ordnet vorhandener IAP-Slaves Kommunikationskanäle und Bandbreite zu. Falls alle Slaves anschließend eine positive Konfigurationsmeldung an den Master senden,

wechselt dieser in den Zustand ACTIVE. Dort sendet er ein START_NODE-Kommando, das den Wechsel der IAP-Slaves in den Zustand ACTIVE bewirkt.

Im Zustand ACTIVE erfolgt die zyklische Kommunikation zwischen Master und Slaves. Der Master sendet ein MDT mit den von der Steuerung generierten Sollwerten und die Slaves senden jeweils ein DDT mit entsprechenden Istwerten. Vom Zustand ACTIVE kann durch die Einwirkung des Anwenders (Initiieren eines STOP_NODE-Kommandos) in den Zustand STOP gewechselt werden. Dort wird die zyklische Kommunikation angehalten und kann erst wieder durch ein START_NODE-Kommando reaktiviert werden. Ein Wechsel in den Zustand ERROR erfolgt, wenn ein Fehler innerhalb der Zustände CONFIG, ACTIVE und STOP auftritt. Dieser Zustand kann erst wieder bei einem Reset (Node- oder Bus-Reset) verlassen werden.

Für eine erfolgreiche IAP-Konfiguration muss der Anwender folgende Systemparameter in einer xml-basierten Konfigurationsdatei spezifizieren:

- Kommunikationsmodus: legt fest, ob asynchrone oder isochrone Kanäle für die Übertragung von MDT und DDT eingesetzt werden
- Taktfrequenz: legt die Frequenz der zyklischen Kommunikation fest (500 bis 8.000 Hz)
- Für jeden Slave
    - Systemweit eindeutige Identifikationsnummer (NodeID)
    - Spezifikation der Sollwerte (Namen und Größe in Byte)
    - Spezifikation der Istwerte (Namen und Größe in Byte)

2.5 IEEE1394 Standard (FireWire)

Die Auswahl des IEEE1394 Standards im SFB562 als Kommunikationsmedium zwischen dem Steuerungsrechner und den KKs war das Ergebnis einer im Rahmen der Arbeit von Beckmann [Bec01] durchgeführten Untersuchung und Bewertung von Hochgeschwindigkeits-Kommunikationssystemen. Auf ihre Eignung wurden folgende Bussysteme untersucht: SERCOS [SER07], MACRO [MAC06], USB [EK01], IEEE1394 [IEE08] und Fibre Channel [FCI10]. Als Bewertungskriterien galten unter anderem die Übertragungsbandbreite, die erreichbare Zykluszeit, die Synchronisationsgenauigkeit, die verfügbaren Kommunikationskanäle und die Unterstützung von Querdatenverkehr.

Zum Entwicklungszeitpunkt der SFB562-Kommunikations-Infrastruktur waren lediglich Integrierte Schaltkreise (IC) des IEEE1394 A-Standards auf dem Markt erhältlich. Inzwischen sind auch IC des B-Standards auf dem Markt weit verbreitet und die letzte Version der IEEE1394 Spezifikation (IEEE Std. 1394-2008) wurde publiziert. Der IEEE1394 A-Standard unterstützt eine paketorientierte Datenübertragung mit Datenraten bis zu 400 Mbit/s bei maximaler Entfernung von 4,5 m zwischen zwei Teilnehmern. Der IEEE1394-Standard unterscheidet zwischen dem isochronen und asynchronen Übertragungsmodus. Er erlaubt den

Anschluss von bis zu 63 Teilnehmern pro Bussegment bei maximal 1023 nutzbaren Bussegmenten.

Für die Integration des IEEE1394 Standards gab es zum Entwicklungszeitpunkt der ursprünglichen SFB562-Kommunikations-Infrastruktur zwar kommerziell verfügbare Hardware-Komponenten (PCI-Einsteckkarten, DSP-basierende Hardwaremodule), aber die dazugehörigen Soft- und Firmware konnten nicht zuletzt wegen ihres allgemeinen Ansatzes die notwendigen Leistungsanforderungen innerhalb des SFB562 nicht erfüllen. Daher wurde eine dedizierte auf Performanz optimierte IEEE1394-Treibersoftware sowohl für den PC als auch für das DSP-Modul entwickelt. Die Treibersoftware wurde als eigenständiger Applikationsprozess entwickelt [Koh07], der durch die weitgehend realisierte Überbrückung der Kernel-Funktionen kleine Übertragungslatenz erzielen kann. Sie bietet einen direkten und deterministischen Zugriff auf schnelle Funktionen der FireWire-Hardware.

2.6 Integration von Roboter-Hardware-Komponenten

Die informationstechnische Integration von Roboter-Hardware-Komponenten in die Kommunikations-Infrastruktur erfolgt über die KKs. Sowohl Aktoren (Roboter-Antriebe) als auch Sensordaten (digital und analog) lassen sich auf diese Weise in dem zyklischen Steuerungszyklus erfassen. Abbildung 2-5 stellt den Aufbau eines KK dar. Es besteht aus drei Hauptmodulen: einem IEEE1394-Modul, einem DSP-Modul und einem Interface-Modul. Diese drei Module stehen als Platinenstapel (Abbildung 2-6) für Integrationszwecke zur Verfügung.

Das IEEE1394-Modul dient dem Anschluss der KK an das IEEE1394-Kommunikationsmedium. Es besteht hauptsächlich aus den kommerziell erhältlichen Standard-ICs PHY[6] und LLC[7], die für eine einwandfreie Abwicklung asynchroner und isochroner Transaktionen sorgen. Auf dem DSP-Modul erfolgt die Ausführung der wesentlichen Software-Komponenten des KK. Neben der Treibersoftware und einer Slave-Instanz des IAP-Protokolls lässt sich die anwenderspezifische Applikation implementieren. Letztere greift auf Roboterkomponenten über ein dediziertes Hardware-Interface-Modul zu, das je nach Typ der Roboterkomponente aus einem DPRAM[8] (bei Antrieben) oder AD/DA-Umsetzer (bei analogen und digitalen Signalen) besteht.

[6] Physical layer controller

[7] Link Layer Controller

[8] Dual Port Random Access Memory

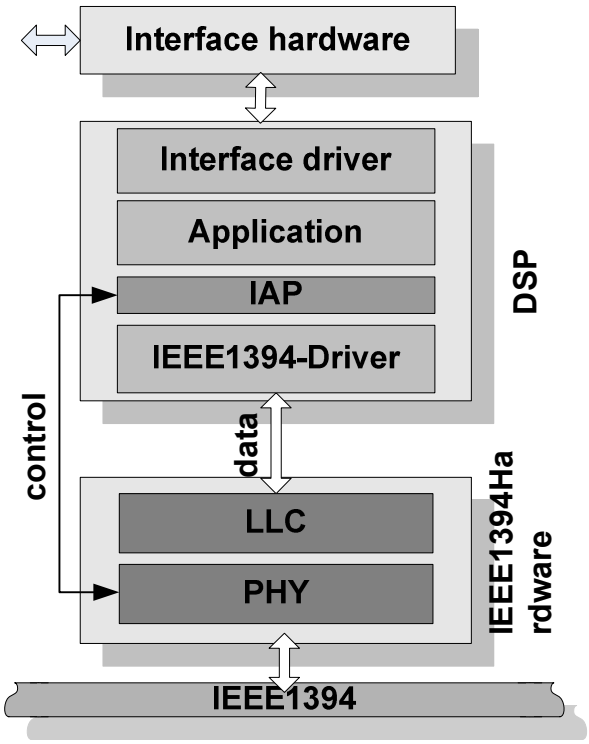

Abbildung 2-5: Schematischer Aufbau eines Hardware-Kommunikationsmoduls (KK)

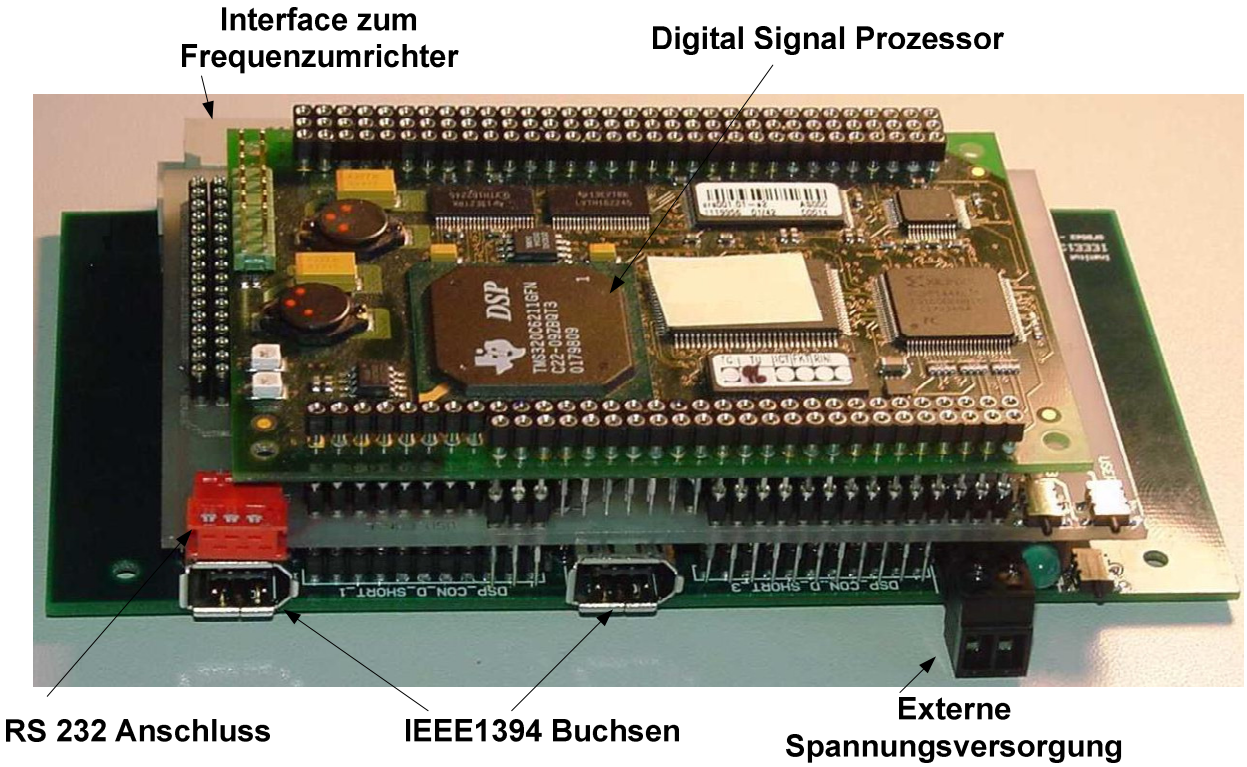

Abbildung 2-6: Exemplarisches Hardware-Kommunikationsmodul (KK)

2.7 Einsatz im RCA562

Das im Rahmen des SFB562 entwickelte neuartige Steuerungssystem RCA562 basiert auf dem von Mason [Mas81] formulierten Task-Frame-Formalismus (TFF), der einer aufgabenorientierten Beschreibung von Handhabungs- und Montagevorgängen dient. Die abstrakten „High-Level“-Befehle, die durch den TFF spezifiziert werden, ermöglichen eine intuitive Programmierung von Roboteraufgaben. Hierbei muss die Steuerungsarchitektur dafür sorgen, dass diese abstrakten Befehle in einfache Steuerungsparameter umgewandelt werden. Im TFF-Kontext werden die „High-Level“-Befehle, die als atomare Roboteraktionen zu interpretieren sind, als Manipulationsprimitive bezeichnet. Durch deren geeignete Verkettung lässt sich eine Vielfalt komplexer Roboteraufgaben auf einfache Weise beschreiben und realisieren. Für die Programmierung von Parallelrobotern wurden in [Maa09] auf bewegte Bezugsysteme erweiterte Manipulationsprimitive beschrieben, die aus folgenden Elementen bestehen:

- Frei wählbares Bezugskoordinatensystem
- Beschreibung der Bewegungsaufgabe
- Transitionsbedingung
- Rückgabewerte

Innerhalb des Manipulationsprimitives lässt sich für jeden Freiheitsgrad im kartesischen Koordinatensystem ein eigener Bewegungsalgorithmus spezifizieren. Die Beschreibung der Bewegungsaufgabe stellt das zentrale Element des Manipulationsprimitives dar. Dort lassen sich hybride Regelungsaufgaben parametrisieren, die das gleichzeitige Regeln von geometrischen Größen und Interaktionsgrößen ermöglichen, die während der Robotermanipulation auftreten [BS04]. Die im Manipulationsprimitive anzugebende Transitionsbedingung legt fest, wann die Ausführung derselben beendet werden soll.

2.7.1 Steuerungssoftware

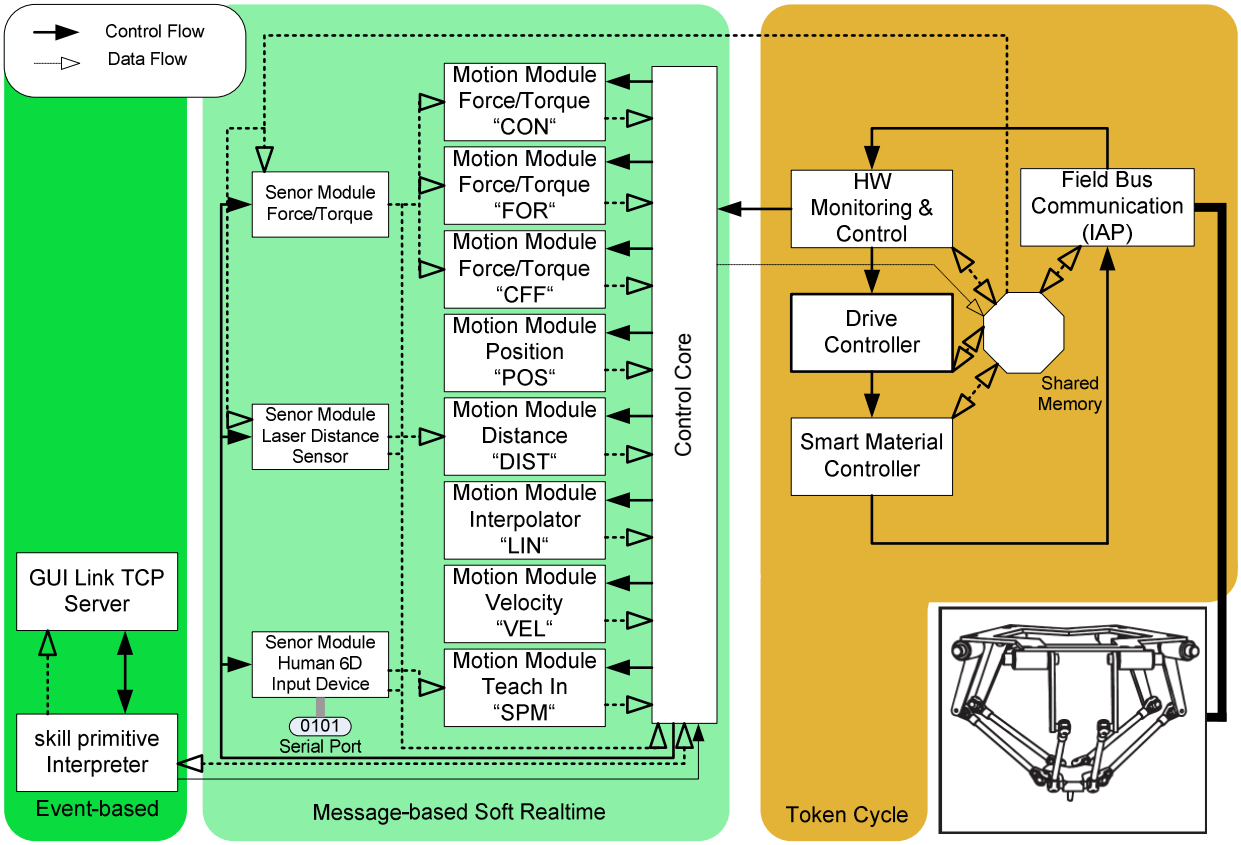

Abbildung 2-7: 3-Schichten-Design der Steuerungsarchitektur [DMM08]

Abbildung 2-7 stellt eine Übersicht der umgesetzten Steuerungsarchitektur dar. Sie besteht aus drei vertikalen Schichten. In der rechten Schicht werden die harten Echtzeit-Steuerungsprozesse dargestellt. Sie werden innerhalb des von MiRPA-X bereitgestellten Token-Passing-Kontexts in einer sequenziellen und deterministischen Weise ausgeführt. Im Token-Zyklus ist der IAP-Prozess integriert, der für die Steuerungssoftware als Synchronisierungsquelle dient und den Austausch von Soll- und Istwerten zwischen der Steuerung und den Sensor- und Aktoreinheiten des Roboters zyklisch koordiniert.

Zu Beginn eines Steuerungszyklus überträgt der IAP-Prozess die über dem IEEE1394-Bus empfangenen Sensordaten zu einem zuvor vereinbarten Shared-Memory-Bereich und gibt anschließend den Token frei. Der MiRPA-X-Scheduler leitet daraufhin den Token an den nächsten Prozess im Token-Zyklus, dem „Hardware Monitoring & Control"-Prozess (HMC), weiter. Der HMC-Prozess ist für die Aktivierung und Abschaltung von Kontroll- und Überwachungsfunktionalitäten des Roboters zuständig. Außerdem aktiviert er nach einem einstellbaren Zyklusverhältnis den in der mittleren Schicht dargestellten Steuerungskern (Control Core). Nachdem der HMC-Prozess den Token freigegeben hat, wird er anschließend an zwei weitere Prozesse weitergeleitet, die innerhalb roboterspezifischer Regelungsalgorithmen neue

Sollwerte berechnen: die Antriebsregelung und die adaptronische Regelung. Letztere wird bei Parallelrobotern angewendet, um durch hohe Verfahrgeschwindigkeit induzierte Strukturschwingungen zu unterdrücken [SMR+04]. Anschließend erhält der IAP-Prozess den Token erneut und übermittelt die generierten Sollwerte an die Roboterantriebe via IEEE1394-Bus. Danach behält er den Token solange, bis ein neuer Zyklus mit dem Empfang von neuen Sensordaten (Istwerten) beginnt.

In der mittleren Schicht erfolgt die Trajektorienberechnung in kartesischen Koordinaten. Dafür wird das aktuelle Manipulationsprimitive durch das Zusammenspiel des Steuerungskerns mit den Sensor- und Bewegungsmodulen (motion module) interpretiert und in einen Satz von Roboterposen umgewandelt. Die Generierung der Trajektorie in kartesischen Koordinaten hat den Vorteil, dass sie durch die Beschreibung der End-Effektor-Bewegung unabhängig von der kinematischen Struktur eines bestimmten Roboters ist. Innerhalb dieser Schicht findet eine nachrichtenbasierte Kommunikation zwischen den beteiligten Prozessen statt. Nachdem der Steuerungskern durch den HMC-Prozess aktiviert wurde, triggert er durch den Versand einer entsprechenden COMMAND-Nachricht die Sensormodule an. Letztere lesen die Sensordaten aus dem Shared-Memory-Bereich aus, führen Signalverarbeitungsalgorithmen (z.B. Filterung und Koordinatentransformationen in den Task-Frame) aus und leiten letztlich die verarbeiteten Daten an die Bewegungsmodule weiter. Die Bewegungsmodule realisieren die Trajektorienberechnung. Innerhalb eines Steuerungszyklus aktiviert der Steuerungskern nur die Bewegungsmodule, die für die Durchführung des aktuellen Manipulationsprimitives erforderlich sind. Die Aktivierung erfolgt über eine COMMAND-Nachricht unter Verwendung des Namensdienstes von MiRPA-X. Nach einer konfigurierbaren Anzahl von Token-Zyklen stellt der Steuerungskern die von den einzelnen Bewegungsmodulen generierten Daten zu einem gültigen Satz von kartesischen Trajektoriendaten zusammen. Anschließend werden die Trajektoriendaten an den Antriebsregler über einen festgelegten Shared-Memory-Bereich weitergeleitet.

In der linken Schicht werden der Manipulationsprimitiv–Interpreter und der GUI-Server dargestellt. Der GUI-Server ist über TCP/IP mit einem auf Windows-Rechner laufenden graphischen Roboterprogrammierungs-Interface verbunden. Beide Module, deren Ausführung keine Echtzeitanforderung im zyklischen Betrieb aufweist, kommunizieren ausschließlich über MiRPA-X-Nachrichten.

2.7.2 Sensormodule

Die Schnittstelle zu Sensordaten lässt sich durch ein Datenobjekt definieren, das neben dem tatsächlichen Sensorwert zusätzlich zwei Statusvariablen beinhält. Die Statusvariablen geben jeweils an, ob Sensorwerte für die Bearbeitung des aktuellen Manipulationsprimitives benötigt werden oder bereitgestellte Sensorwerte gültig sind. Bei der Ausführung eines neuen Manipulationsprimitives ermittelt der Steuerungskern, welche Sensorwerte benötigt werden, und stellt eine entsprechende Anfrage in dem Sensordatenobjekt. Die Aktivierung von

Sensormodulen im zyklischen Betrieb erfolgt über MiRPA-X-Nachrichten. Nach der Aktivierung überprüfen die Sensormodule die entsprechenden Sensordatenobjekte auf einen Berechnungsauftrag. Falls Sensorwerte angefordert sind, dann führen sie den entsprechenden Algorithmus zur Sensorsignalverarbeitung aus. Anschließend setzen sie entsprechend dem Ergebnis des Signalverarbeitungsalgorithmus die Gültigkeit der Sensordaten bestimmende Statusvariable im Sensordatenobjekt. Um den Aufbau einer selbstkonfigurierbaren Steuerung zu unterstützen, legt jedes Sensormodul ein Applikationsprofil an, das auf Anfrage veröffentlicht werden kann. Die enthaltenen Profilinformationen beschreiben unter anderem eine Liste der angebotenen Sensordatenobjekte.

2.7.3 Bewegungsmodule (Motion Module)

Die Bewegungsmodule kapseln Bewegungsalgorithmen, die innerhalb eines Manipulationsprimitives jeweils für die Bewegungssteuerung eines einzelnen kartesischen Freiheitsgrades spezifiziert werden können. Bei der Bearbeitung eines neuen Manipulationsprimitives werden die Steuerungsinformationen, die die auszuführenden Bewegungsalgorithmen inklusive ihrer Parameter beschreiben, der im Primitiven enthaltenen Bewegungsaufgabe entnommen und der zyklischen Aktivität bereitgestellt. Im Gegensatz zu Sensormodulen werden die Bewegungsmodule zur Ausführung nicht global aktiviert, sondern gezielt, je nach dem, ob sie für die Verarbeitung des aktuellen Manipulationsprimitives benötigt werden. Der Datenaustausch mit dem Steuerungskern erfolgt, ähnlich wie bei Sensormodulen, über festgelegte Datenobjekte. Um den Aufbau einer selbstkonfigurierbaren Steuerung zu unterstützen, legt jedes Bewegungsmodul ebenfalls ein Applikationsprofil an. Die bereitgestellten Profilinformationen beinhalten den Namen des Algorithmus sowie eine Liste benötigter Sensordatenobjekte.

2.7.4 Steuerungskern (Control Core)

Der Steuerungskern (Abbildung 2-7) ist das zentrale Element der RCA562-Architektur. Um eine selbstkonfigurierende Steuerung zu realisieren, sendet der Steuerungskern nach dem Start eine globale Profilanfrage an alle vorhandenen Sensor- and Bewegungsmodule mittels einer COMMAND-Nachricht. Diese Module reagieren auf die Anfrage mit der Veröffentlichung ihrer Profilinformationen. Anhand dieser Informationen kann der Steuerungskern zur Laufzeit ermitteln, welche Module für die Ausführung des aktuellen Manipulationsprimitives aktiviert werden müssen. Dank dieses Mechanismus können die gestarteten Modul-Prozesse selbständig und automatisch zu einer funktionierenden Steuerung zusammenfinden. Der Einsatz der Kommunikations-Middleware MiRPA-X erlaubt es, dynamische Kommunikationswege zwischen den Steuerungsmodulen zu realisieren. Auf diese Weise lassen sich immer, abhängig von der aktuellen Aufgabe, optimal geeignete Regelalgorithmen anwenden. Diese Prozessabhängige Softwareumschaltung ist z.B. bei den in Kapitel 2.1 genannten Systemen

wegen deren teilweise fest vorgegebenen Kommunikationsstruktur und Kommunikationslatenz nicht möglich.

2.8 Grenze der Kommunikations-Infrastruktur

Im SFB562 konnte die entwickelte Kommunikations-Infrastruktur umgesetzt und deren Funktionsfähigkeit hauptsächlich an zwei kinematischen Strukturen (Abbildung 2-8) nachgewiesen werden. Mit der aufgebauten Infrastruktur lassen sich aufgrund des hohen zyklischen Kommunikationaufwands jedoch nur Steuerungszyklen mit einer Frequenz von maximal 1 kHz realisieren.

Das dynamische Potenzial von Parallelrobotern lässt sich aufgrund dieser technischen Gegebenheit nicht voll ausschöpfen. Die möglichen hohen Verfahrgeschwindigkeiten erfordern kürzere Steuerungs- und Regelungszyklen, um eventuell auftretende Störgrößen und Regelabweichungen, die zur Zerstörung der mechanischen Struktur und im schlimmsten Fall sogar zur Bedrohung menschliches Lebens führen können, rechzeitig aufzufangen und steuerungstechnisch zu korrigieren. Im Allgemein gilt, je kürzer der Steuerungszyklus, umso besser die Regelgüte. Im SFB562 ist demzufolge eine Erhöhung des realisierbaren Steuerungszyklus auf mindestens zwei kHz erwünscht. Die dafür notwendigen Systemoptimierungsmaßnahmen werden in den nächsten Kapiteln dieser Arbeit behandelt.

Abbildung 2-8: SFB562-Demontratoren – links TRIGLIDE, rechts HEXA

3 Optimierungspotenzial der Kommunikations-Infrastruktur

Die im vorigen Kapitel vorgestellte Kommunikations-Infrastruktur bot eine in vieler Hinsicht sehr gute Plattform für die Umsetzung der von Maaß in [Maa09] entwickelten Steuerungskonzepte. Dennoch ergeben sich aus gestiegenen Anforderungen der Anwendung auch Optimierungsanforderungen für die Performanz und die Funktionsweise einiger bereitgestellter Mechanismen. Bisher konnte der Steuerungskreis lediglich mit einer maximalen Frequenz von 1 kHz betrieben werden. Die Gründe dafür lagen teilweise in der Entwurfsebene des Software-Frameworks, das aufgrund der eingeschränkten Einsatzflexibilität einiger Kommunikationsmechanismen die Implementierung der obengenannten Steuerungskonzepte nur über hinsichtlich der Performanz kostenaufwendige Umwege zuließ. Weitere Ursachen der begrenzten Steuerungsfrequenz lagen in den Datenverarbeitungs- und Datenaustauschmechanismen innerhalb der auf DSP basierenden Kommunikationsmodule.

Die Optimierung der Kommunikations-Infrastruktur ist eines der Hauptthemen dieser Arbeit. In diesem Kapitel werden die einzelnen Funktionalitäten in Soft- und Hardware innerhalb der Kommunikations-Infrastruktur identifiziert, die ein Optimierungspotenzial aufweisen, um beispielsweise eine Steigerung der Steuerungsfrequenz zu erreichen. Untersucht werden die Middleware MiRPA-X, das IAP, der eingesetzte IEEE1394 Standard und die Hard- und Software-Komponenten der Kommunikationsmodule.

3.1 MiRPA-X

Aus praktischem Einsatz und weitergehenden Untersuchungen hat sich gezeigt, dass der strukturelle Aufbau einiger MiRPA-X Funktionalitäten die Umsetzung mancher Software-Design-Konstrukte innerhalb der Steuerungsarchitektur erschwert und sich somit das gewünschte Verhalten nur über Umwege und Performanzeinbußen realisieren lässt. In diesem Abschnitt werden solche Funktionalitäten identifiziert und Optimierungsmöglichkeiten diskutiert.

3.1.1 Client/Server Applikationsmodell

In der MIRPA-X-Umgebung werden Applikationsprozessen bei der Registrierung entweder Client- oder Server-Eigenschaften zugeordnet, dabei darf ein Server-Prozess keinen Nachrichtentransfer initiieren. Diese designbedingte Einschränkung erschwert den flexiblen, hierarchischen und performanzoptimierten Aufbau der Steuerungsarchitektur (Abschnitt 2.7) erheblich. Das Design der Steuerungssoftware sieht vor, dass Server-Prozesse wegen hoher Komplexität ihre Funktionen auf weitere sekundäre Server verteilen müssen. Da ein Server-Prozess keinen Nachrichtentransfer initiieren darf, muss ein zusätzlicher Client-Prozess implementiert werden, der die Funktionsverteilung des Servers vornimmt. Um eine einwandfreie Funktionsweise zu gewährleisten, müssen die zwei Prozesse zusätzlich miteinander

synchronisiert werden (Abbildung 3-1a). Die zusätzliche Synchronisierung erhöht den Software-Entwicklungsaufwand und reduziert gleichzeitig die erreichbare Performanz der Anwendung.

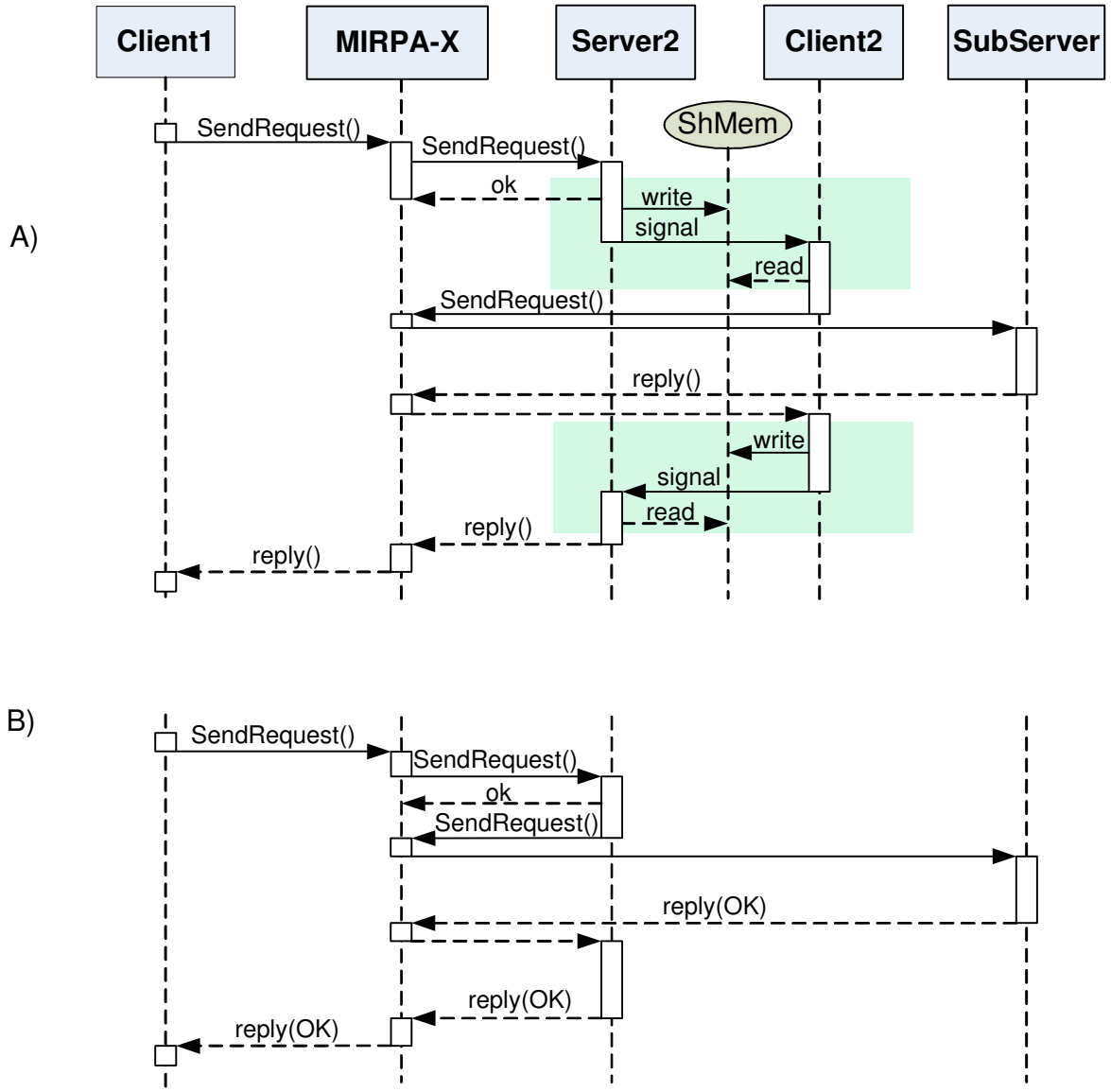

Abbildung 3-1: Exemplarisches Anwendungs-Design. a) strenges Client und Server Design mit Applikationsprozessen exklusiv als Client oder Server, farbig gekennzeichnete Bereiche stellen zusätzlichen Entwicklungsaufwand und Performanzeinbuße dar. b) modifiziertes Client und Server Design mit Applikationsprozessen sowohl Client als Server

Um eine bessere Performanz zu ermöglichen und den Entwicklungsprozess zu erleichtern, muss das Client/Server-Modell so modifiziert werden, das jeder Applikationsprozess sowohl Client- als auch Server-Eigenschaften besitzen darf. Auf dieser Weise dürfte auch ein Server Nachrichten initiieren. Entsprechend dieser strukturellen MIRPA-X-Änderung würde sich das in Abbildung 3-1b dargestellte und optimierte Design ergeben. Der Datenaustausch zwischen

Client und Server über den gemeinsamen Speicher (ShMem) und die der Synchronisationsaufwand würde entfallen. Die würde zu einer Reduzierung der Kommunikationszeit bis zu 30 % führen.

Die strukturelle Änderung des Client/Server-Modells bringt zwar mehr Performanz und Flexibilität bei der Softwareentwicklung, hebt aber zugleich die bisher von MIRPA-X gewährleistete Deadlock-Freiheit auf. Es müssen also zusätzliche Mechanismen zur aktiven Deadlock-Detektion und Verhinderung in die Middleware integriert werden. Auf die technischen Details bezüglich der Änderung am Client/Server-Modell und der Mechanismen zur Deadlock-Detektion wird im Abschnitt 4.1 eingegangen.

3.1.2 Synchrone Kommunikation ohne User-Daten

Der synchrone Kommunikationsmechanismus wird unter MIRPA-X mittels REQUEST-Nachrichten realisiert. Er sieht vor, dass ein Client-Prozess eine Anfrage an einen Server-Prozess stellt und blockiert, bis er die Antwort des Servers bekommt. MIRPA-X gestattet es aber dem Client-Prozess bisher nicht, umfangreiche Applikationsdaten mit der Anfrage an den Server zu übertragen. Lediglich ein wort-breites Feld wird bei der Übermittlung von REQUEST-Nachrichten für Applikationsdaten bereitgestellt. Umfangreiche Applikationsdaten müssen demzufolge asynchron über eine zusätzliche COMMAND-Nachricht vorher an den Server gesendet werden (Abbildung 3-2).

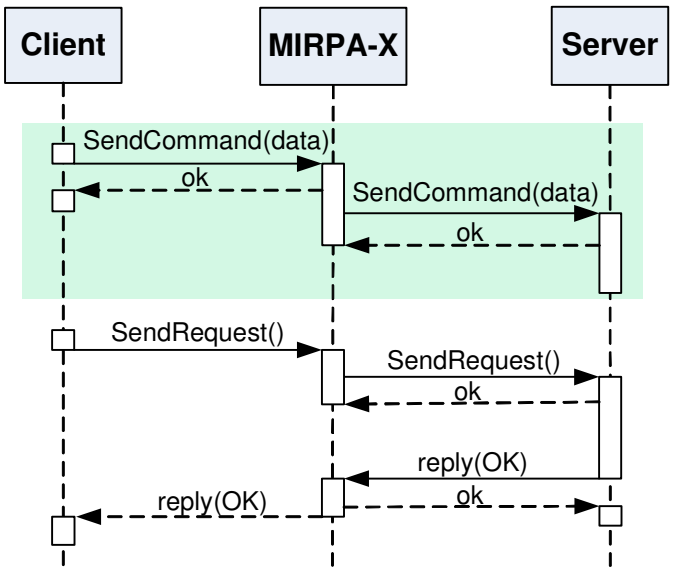

Abbildung 3-2: Performanzeinbuße bei Übermittlung von User-Daten über eine zusätzliche COMMAND-Nachricht.

Durch die zusätzliche COMMAND-Nachricht wird die Kommunikationszeit um mehr als 10 µsec erhöht und die Performanz der Anwenderapplikation reduziert (farbig markierter Bereich in

Abbildung 3-2). Die designbedingte Sperrung der Übertragung von Applikationsdaten mit den synchronen REQUEST-Nachrichten hat eine Performanzeinbuße zur Folge und sollte im Zuge der Systemoptimierung aufgehoben werden. Näher wird im Abschnitt 4.2 darauf eingegangen.

3.1.3 Konfigurations- und Kommunikationsnachrichten

Sämtliche nachrichtenbasierte Transaktionen werden von MIRPA-X in einer sequentiellen Weise nach dem FIFO-Scheduling-Modell verwaltet. Grundsächlich lassen sich die Transaktionen in zwei Klassen einteilen: a) die Konfigurationsnachrichten, die der An- und Abmeldung von Ressourcen und Applikationsprozessen dienen und b) die IPC[9]-Nachrichten, die Echtzeit-Kommunikation zwischen Applikationsprozessen darstellen und in dem MiRPA-X Kontext durch synchrone REQUEST und asynchrone COMMAND vertreten sind.

Das Design der Middleware sieht einen einzigen Thread für die Verwaltung von IPC- und Konfigurationsnachrichten vor. Dabei werden beide Nachrichtenklassen gleichwertig behandelt. Die Begründung für das *single threaded* Design liegt in den angestrebten kurzen Abarbeitungszeiten pro Echtzeit-Kommunikationsnachricht. Andere Middleware [FKK+07] setzen auf eine *multi-threaded* Lösung, weisen aber eine geringere Performanz auf, die zum Teil durch den zusätzlichen Kontextwechsel bedingt ist. Der *single threaded* Ansatz hat einen Nachteil, denn er birgt ein Konfliktpotenzial zwischen Konfigurations- und IPC-Nachrichten. Konfigurationsnachrichten können unter Umständen die Bearbeitung von IPC-Nachrichten verzögern. Ein solches Szenario wurde bei der Entwicklung der RCA562 Architektur dokumentiert. Die Graphik in Abbildung 3-3 zeigt eine Aufnahme der Systemausführung in Echtzeit, die mittels des Monitoring-Tools der instrumentierten Version von *QNX Neutrino Microkernel* [QNX09] gemacht wurde. Sie zeigt die zeitlichen Interaktionen zwischen den ausgeführten Prozessen und Threads. Im markierten Bereich wird die Abmeldung des Prozesses „triglidetest“ ausgeführt. Dafür wird eine entsprechende Konfigurationsnachricht an den Middleware-Prozess „observ“ geschickt. Bei der Abarbeitung dieser Konfigurationsnachricht ist der „observ“ Prozess im ungünstigen Fall ca. 3 ms ausgelastet. Während dieser Zeit kann keine IPC-Nachricht von der Middleware bearbeitet werden. In diesem Fall hatte dieses Szenario die Verletzung von Echtzeitbedingungen und den Absturz der RCA562 Steuerung zur Folge.

[9] Inter Process Communication

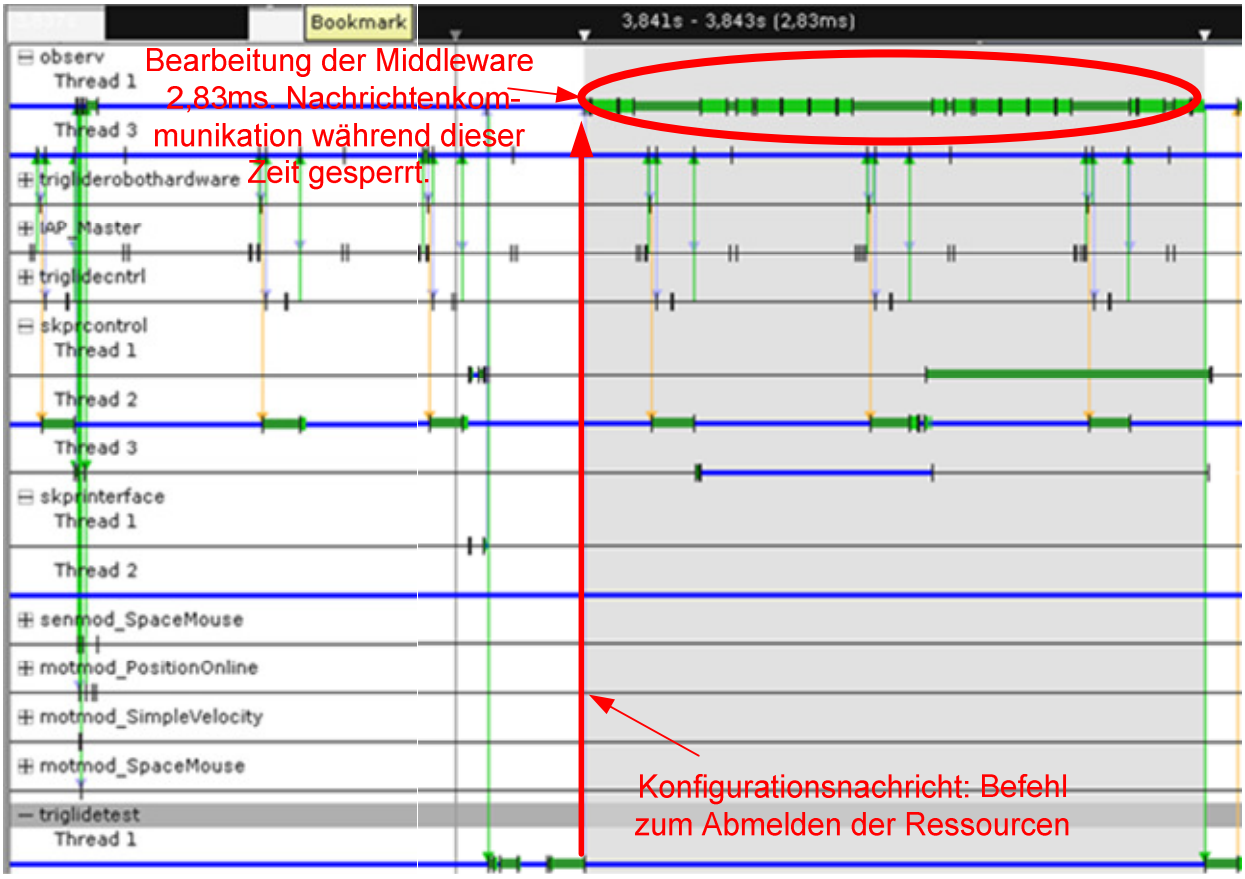

Abbildung 3-3: Trace-Aufnahme der Systemausführung in Echtzeit: Konfigurationsnachricht behindert IPC-Nachricht

Um dieses unerwünschte Verhalten zu umgehen, wird beim Einsatz der Middleware zwischen einer Konfigurationsphase, in der sämtliche Applikationsprozesse gestartet und Ressourcen konfiguriert werden und dem laufenden Betrieb, während dessen ausschließlich die IPC-Kommunikation zugelassen ist, unterschieden. Diese Einschränkung führt natürlich zu einer weiteren Verringerung der Flexibilität und erschwert das reibungslose Updaten von Software-Komponenten im laufenden Betrieb. Eine bessere Lösung bietet die Realisierung von Nebenläufigkeit bei der Bearbeitung von Konfigurations- und IPC-Nachrichten. Hierbei soll die Middleware zwei Threads mit unterschiedlichen Prioritäten zur Verfügung stellen, die die zwei Nachrichtenklassen separat bearbeiten. Die Einführung der Nebenläufigkeit wird Abschnitt 4.3 behandelt.

3.1.4 Token-Scheduling

Um die Token-Threads miteinander zu synchronisieren, die über definierte Shared-Memory-Bereiche auf hoher Priorität Daten zyklisch austauschen, realisiert der MIRPA-X-Scheduler ein serielles Token-Scheduling. Dabei aktiviert er die Token-Threads nacheinander entsprechend einer vom Anwender einstellbaren Reihenfolge. Wegen des seriellen Token-Scheduling kann

aber das gesamte Rechenpotenzial nicht genutzt werden, das durch Mehrkernprozessoren zur Verfügung gestellt wird. Denn Token-Threads, die keine Datenabhängigkeiten untereinander aufweisen und auf einem Mehrkernprozessor theoretisch nebenläufig ausgeführt werden können, müssen vom Scheduler auch nacheinander aktiviert werden. Um Token-Threads auf einer Mehrkernprozessor-Plattform parallel ausführen zu können, muss eine entsprechende funktionelle Änderung des Token-Schedulers vorgenommen werden, die eine gleichzeitige Aktivierung mehrerer Token-Threads bei Erhalt der Datenintegrität unterstützt. Im Abschnitt 4.4 wird die Realisierung der neuen Scheduler-Funktionalität erläutet.

3.2 IAP

Die im IAP-Protokoll implementierte Fehlerbehandlung ist für Roboteranwendungen nicht ausreichend gut implementiert. Die IAP-Spezifikation ermöglicht den Einsatz des IEEE1394 für Aufgaben, die eine hohe Taktsynchronität und Deterministik erfordern. Im aktiven Zustand erfolgt der Datenaustausch zwischen der Steuerung und den Kommunikationsteilnehmern über einen festgelegten zyklischen Kommunikationszyklus, innerhalb dessen MDT- und DDT-Telegrammtransaktionen ausgeführt werden. Am Anfang eines neuen Kommunikationszyklus prüft der IAP-Master, ob sämtliche Teilnehmer ein DDT-Telegramm im vergangenen Kommunikationszyklus gesendet haben. Sollte dies nicht der Fall sein, dann ist ein Transaktionsfehler detektiert. Die Transaktionsfehler können mehrere Ursachen haben: defekter Teilnehmer, EMV-bedingte Übertragungsfehler oder fehlerhafter Telegrammempfang im IAP-Master-Modul. Nach Erkennung des Transaktionsfehlers wechselt der IAP-Master in einen Fehlerzustand und bricht sofort die zyklische Kommunikation mit den Teilnehmern ab. Mit dem Abbruch der zyklischen Kommunikation werden gleichzeitig der Steuerungs- und der Regelungskreis unterbrochen. Daraus folgt, dass die Teilnehmer keine neuen Sollwerte bekommen und die Synchronisation mit den Antriebsverstärkern verlieren. Dann führen die Motoren eine unkontrollierte Notbremsung aus, die im schlimmsten Fall zur Beschädigung der mechanischen Struktur des Roboters führen könnte.

Die aktuelle Fehlerbehandlung im IAP Protokoll stellt also ein Sicherheitsrisiko bei Roboter-anwendungen dar und muss aus diesem Grund verbessert werden. Es müssen außerdem fehlertolerante Mechanismen eingebaut werden, die gelegentlich auftretende Transaktionsfehler so verkraften, dass die zyklische Kommunikation nicht abgebrochen werden muss. Es soll sichergestellt werden, dass die Steuerung bei Transaktionsfehlern benachrichtigt wird und gegebenenfalls kontrollierte Motorbremsung ausführen kann. Im Abschnitt 4.5 wird näher beschrieben, wie die Fehlertoleranz in das IAP Protokoll integriert wurde.

3.3 IEEE1394

Durch den Einsatz des damals verfügbaren IEEE1394a Standards ist die Datenübertragungszeit zwischen dem Steuerungsrechner und den Teilnehmern relativ lang. Die Busarbitrierung des

IEEE1394 A-Standards erzwingt eine Buslatenzzeit von mindestens 15 µs zwischen zwei asynchronen Telegrammen. Bei sechs Antrieben gehen insgesamt 75 µs pro Kommunikationszyklus allein aufgrund dieser Eigenschaft des Bussystems verloren. Außerdem hat die maximale Datenrate von 400 Mbit/s eine zusätzliche Auswirkung auf die Kommunikationszeit: 10 µs werden für die Busübertragungszeit bei Telegrammgröße von 500 Bytes benötigt.

An dieser Stelle ist ein Upgrade auf dem IEEE1394b Standard im Hinblick auf den Performanzgewinn notwendig. Die im IEEE1394b Standard spezifizierte Busarbitrierung BOSS[10] erfolgt parallel zu der Übertragung von asynchronen Telegrammen. Dies hat den Vorteil, dass die Buslatenzzeit bis auf weniger als 2 µs reduziert und die Datenübertragungszeit verkürzt wird (Abbildung 3-4). Außerdem bietet der B-Standard eine höhere Datenrate von 800Mbit/s, die eine Halbierung der Busübertragungszeit pro Datentelegramm ermöglicht. Ein Upgrade auf dem IEEE1394b Standard verkürzt die Datenübertragungszeit und trägt zur Realisierung eines höheren Steuerungszyklus bei.

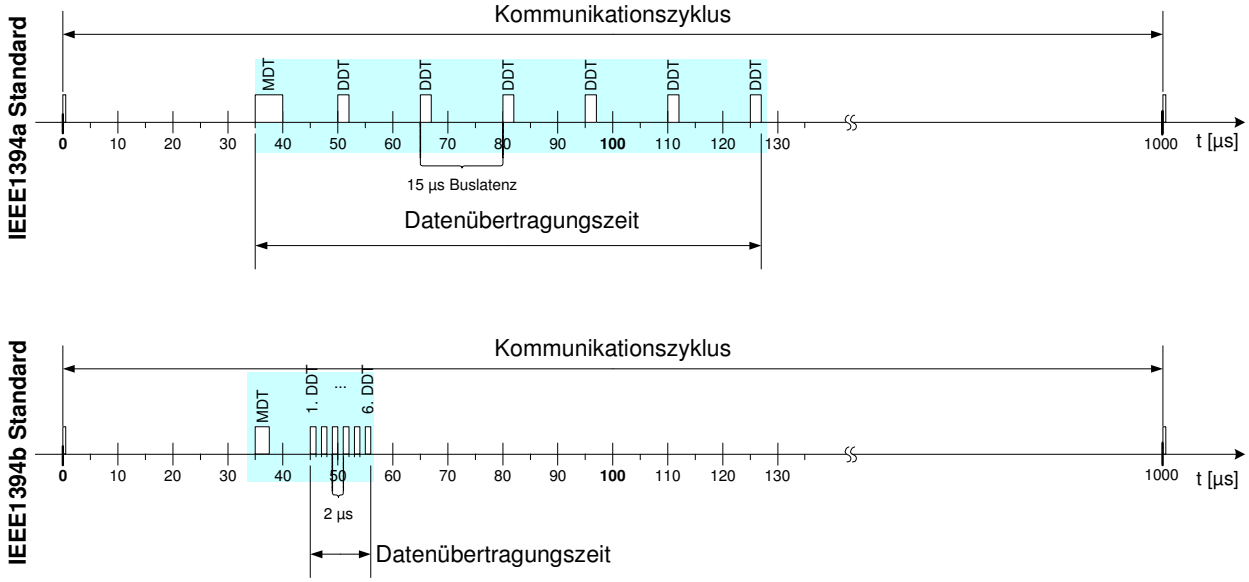

Abbildung 3-4: Datenübertragungszeit – Vergleich zwischen dem IEEE1394a und IEEE1394b Standard

3.4 Kommunikationsmodule

Eine gründliche Analyse des Kommunikationssystems zeigt, dass die Kommunikationsmodule und deren Datenaustauschmechanismus mit den Frequenzumrichtern die größten Optimierungsmöglichkeiten hinsichtlich des Performanzgewinns anbieten. In Abbildung 3-5 wird ein

[10] Bus Owner Supervisor Selector

Kommunikationszyklus von 1 kHz und sechs Antrieben in zwei Zeitsegmente aufgeteilt: die Kommunikationszeit und die Applikationszeit. Die Kommunikationszeit beginnt mit dem Eingang des vom Bussystem generierten CST-Telegramms. Für eine bessere Übersicht sind die CST-Telegramme nur am Anfang und Ende des Zyklus eingezeichnet. Nach Eingang des CST-Telegramms werden die neuen Sollwerte auf dem Steuerungsrechner zusammengestellt und über ein MDT an sämtliche Kommunikationsmodule gesendet. Dieser Vorgang benötigt ca. 30 µs. Nach Empfang des MDT werden die Sollwerte auf den Kommunikationsknoten für den Datenaustausch mit dem Frequenzumrichter aufbereitet. Dabei extrahiert jedes Modul die relevanten Daten aus dem MDT und schreibt sie in dedizierte Speicherzellen auf einem DPRAM, das als Austauschplattform dient. Die Sollwerte-Aufbereitung dauert insgesamt über 200 µs. Anschließend aktiviert die Software den Frequenzumrichter, der den Datenaustausch nach einer vom Hersteller vorgegebenen Zugriffssequenz vornimmt. Er liest die Sollwerte aus dem DPRAM und schreibt später die neuen Istwerte in entsprechende vorgegebene Speicherzellen. Der Datenaustausch stellt im Zyklus eine Totzeit von 250 µs dar. Nach dem Datenaustausch werden die Istwerte mittels DDT an den Steuerungsrechner zurückgeschickt. Die Kommunikationszeit endet mit dem Empfang sämtlicher DDTs auf dem Steuerungsrechner. Innerhalb der nachfolgenden Applikationszeit wird die Steuerungsapplikation ausgeführt und die neuen Sollwerte anhand der vorher eingegangenen Istwerte berechnet.

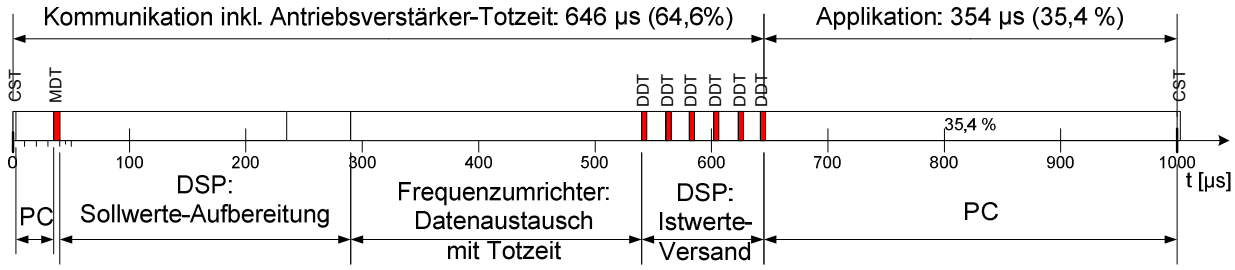

Abbildung 3-5: Aufteilung des Kommunikationszyklus in Rechen- und Kommunikationszeit

In einem Steuerungszyklus von 1 kHz mit sechs Antrieben wird über 646 µs für den Datenaustausch zwischen Steuerungs-PC und den Kommunikationsknoten in Anspruch genommen. Dies entspricht 64,6% der Zykluszeit. Aus diesem Grund kann z.B. kein 2 kHz Steuerungszyklus realisiert werden. Außerdem steht mit 354 µs nicht immer ausreichend Zeit für komplexere und rechenaufwendigere Aufgaben zur Verfügung. Die Ursachen für den langsamen Datenaustausch liegen in der zeitlichen Datenverarbeitung innerhalb der Kommunikationsmodule.

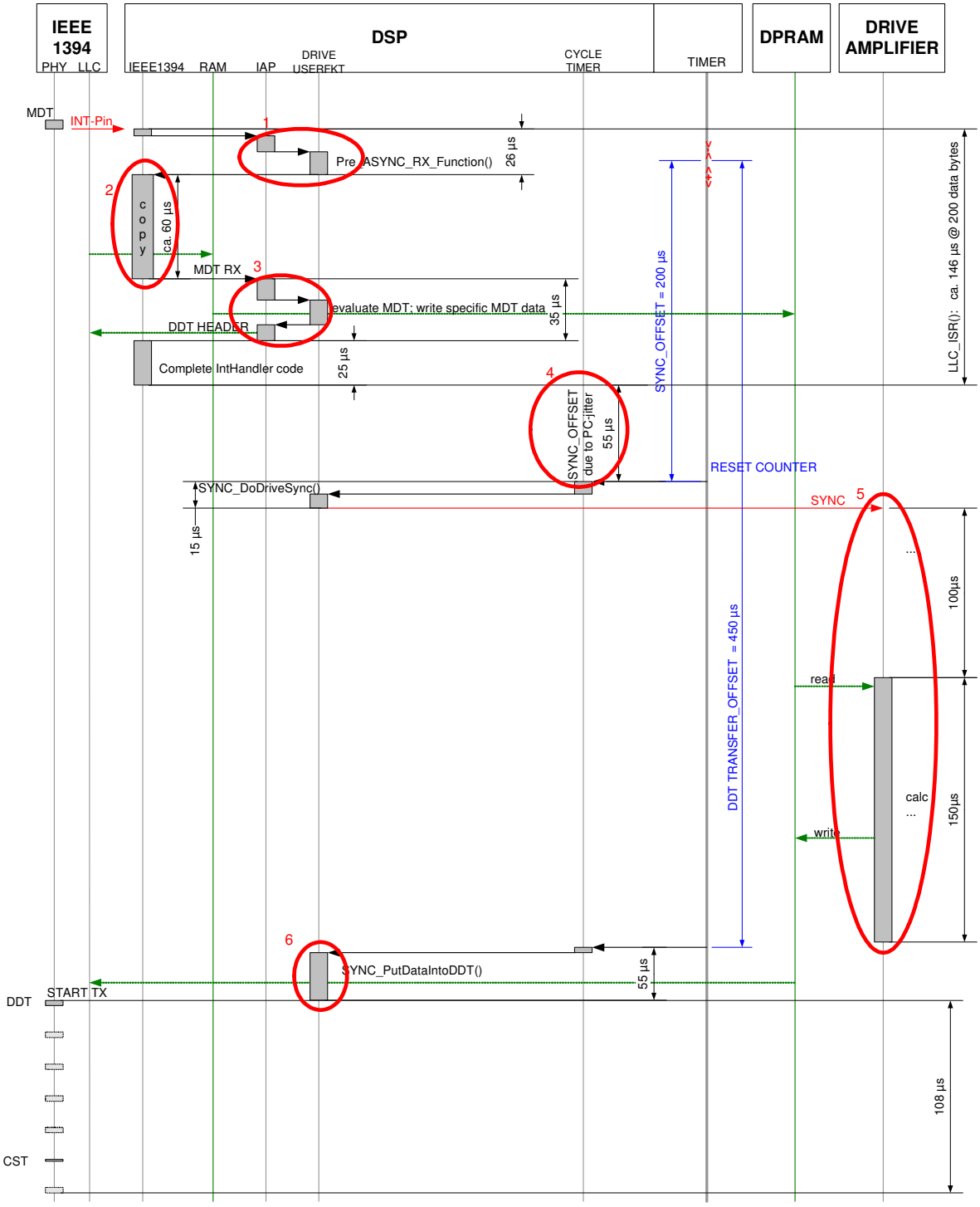

Abbildung 3-6: Detaillierte Beschreibung der Datenverarbeitung innerhalb eines Kommunikationsmoduls

Für eine nähere Analyse der Datenverarbeitung wird das Sequenzdiagramm in Abbildung 3-6 herangezogen. Das Sequenzdiagramm zeigt die verschiedenen Datenverarbeitungsschritte sowie die Software-Interaktionen zwischen den einzelnen Hardware-Komponenten (IEEE1394, DSP, DPRAM und Drive Amplifier) eines Kommunikationsmoduls. Für eine bessere Übersicht

werden die Verarbeitungsschritte in sechs Stufen aufgeteilt (siehe farbig eingekreiste Bereiche in Abbildung 3-6), die im Folgenden beschrieben werden:

- Stufe 1: Synchronisierung der Systemuhren. Dies erfolgt unmittelbar nach dem Empfang des MDT Telegramms. Eine Synchronisierung ist an der Stelle notwendig, weil für den Steuerungsrechner und die Kommunikationsmodule unterschiedliche Systemuhren eingesetzt sind. Im Steuerungsrechner wird die FireWire-inhärente und jitterarme Systemuhr eingesetzt, während die Kommunikationsmodule mit selbstverwalteten Timern arbeiten. Der Einsatz von Timern lässt sich durch die ereignisgesteuerte Arbeitsweise der Software begründen. Mit deren Hilfe lassen sich User-Events (z.B. DDT-Versand, Generierung eines Synchronisationspulses für den Frequenzumrichter) zu vorgegebenen Zeitpunkten in den Softwareablauf integrieren. Für die Synchronisierung wird der FireWire-Clock als Referenz genommen und das Timer-Vergleichsregister entsprechend der Zeitdifferenz beider Uhren angepasst. Die Synchronisierung ist rechenaufwendig, gemessen wurde eine Zeitdauer von 26 µs. Mit einer geschickten Änderung der Softwarestruktur, die die Kopplung sämtlicher User-Events an die FireWire-Uhr berücksichtigt und den Einsatz von zusätzlichen Timern verhindert, könnte diese Zeit gespart und die Datenverarbeitungszeit verkürzt werden.
- Stufe 2: Kopieroperationen. Hierbei wird das empfangene MDT aus dem FIFO des IEEE1394 LLC über eine EMIF[11]-Schnittstelle ausgelesen und in den Speicher geschrieben. Die Dauer des Kopiervorgangs ist von der Größe des Telegramms abgängig; bei 200 Bytes Datengröße dauert es ca. 60 µs. Diese zweite Verarbeitungsstufe kann wegoptimiert werden, wenn ein IEEE1394 LLC mit direktem Speicherzugriff statt mit FIFO-Interface integriert wird, so dass keine zusätzliche Kopieroperation nötig wäre. Ein solches Interface bietet der LLC FireLink Core [Cor07] der Firma Dap Technology [Dap09].
- Stufe 3: Aktualisierung der Sollwerte. An dieser Stelle werden die Sollwerte, die für das jeweilige Kommunikationsmodul relevant sind, aus dem MDT extrahiert und in das externe DPRAM geschrieben. Der Schreibzugriff auf das DPRAM erfolgt byteweise und die Aktualisierung der Sollwerte dauert insgesamt 35 µs. Die Ausführungszeit dieser Verarbeitungsstufe kann dadurch verkürzt werden, dass ein DPRAM Core integriert wird, der 32-bitweise Zugriffe unterstützt. Außerdem ist die Zugriffsgeschwindigkeit beim integrierten Core in der Regel größer als beim externen Chip.
- Stufe 4: Synchronisierung des Frequenzumrichters. Damit der Frequenzumrichter den Datenaustausch vornehmen kann, muss er durch einen Synchronisationsimpuls aktiviert werden. Der Timer-Event zur Impulsgenerierung wird erst 55 µs nach Aktualisierung der Sollwerte im DPRAM ausgelöst. Die Wartezeit von 55 µs gilt als zusätzliche Sicherheitszeit gegen mögliche Jitter des Versandszeitpunkts von MDTs. Dadurch soll

[11] External Memory Interface

erreicht werden, dass Sollwerte immer aktualisiert werden, bevor der Synchronisationsimpuls generiert wird.

- Stufe 5: Datenaustausch mit dem Frequenzumrichter. Die Datenaustauschsequenz ist vom Hersteller festgelegt. 100 µs nach Erhalt des Synchronisationsimpulses liest der Frequenzumrichter die Sollwerte aus dem DPRAM und schreibt nach weiteren 90 µs die Istwerte in das DPRAM. Der Datenaustausch erfolgt innerhalb eines auf dem Frequenzumrichter laufenden 4 kHz zyklischen Tasks. Insgesamt entsteht während des Austausches eine Totzeit von 250 µs, die die Performanz des Systems sehr beeinträchtigt. Eine mit der Wegoptimierung der Totzeit verbundene Änderung der Datenaustauschsequenz ist notwendig.
- Stufe 6: Istwertversand. Der zugehörige Timer-Event wird 450 µs nach Empfang der Sollwerte ausgelöst. Danach werden die Istwerte aus dem DPRAM gelesen und in den IEEE1394 LLC Sendepuffer geschrieben. Die Zusammenstellung des DDTs dauert 55 µs. Eine Performanzoptimierung kann in diesem Fall ebenfalls durch die Integration eines DPRAM Core mit 32-bitweise Schreibzugriffen erreicht werden.

Zusammenfassend weisen die Kommunikationsmodule ein großes Optimierungspotenzial auf. Dies hat mehrere Ursachen: a) den Einsatz von Standard-Komponenten, die die Ausführungsgeschwindigkeit zyklischer Kopiervorgänge (insgesamt 150 µs für Kopieroperationen in einem Zyklus) begrenzen, b) den Einsatz von zwei getrennten Systemuhren und deren notwendiger Synchronisierung (26 µs), c) die eingebaute Sicherheitszeit (55 µs) gegen mögliche Jittern und d) die durch den Datenaustausch mit dem Frequenzumrichter bedingte Totzeit von 250 µs. Um eine bessere Performanz zu erzielen, ist ein neues Design in Hard- und Software notwendig. Das neue Hardware-Design soll auf Basis der FPGA-Technologie erfolgen; IEEE1394 LLC und DPRAM sollen als IP Core in ein FPGA integriert werden, um Datenzugriffe zu beschleunigen und Kopieroperationen zu reduzieren. Das neue Software-Design soll mit einer einzigen Systemuhr ausgestattet sein. Durch die Kopplung des Synchronisationsimpulses an das MDT und die Änderung der Datenaustauschsequenz mit dem Frequenzumrichter soll zusätzliche Zeit gewonnen werden. Details zu den Optimierungen an den Kommunikationsknoten werden im Kapitel 5 erläutet.

3.5 Zusammenfassung

In diesem Kapitel ist die Kommunikations-Infrastruktur untersucht und das Optimierungspotenzial identifiziert worden. Außerdem wurde bei jeder Funktionalität die entsprechende Optimierungsmaßnahme und deren Einfluss auf die Systemperformanz beschrieben.

In der Middleware MIRPA-X wurde das Client-Server Modell, das die Registrierung von Applikationsprozessen ausschließlich entweder als Client oder Server zulässt, als Entwurfmuster identifiziert, das einen hierarchischen und auf Performanz optimierten Aufbau der

Steuerungsarchitektur erschwert. Als Verbesserung sollte die Funktionsweise dieses Kommunikationsmodells so geändert werden, dass Applikationsprozesse gleichzeitig als Client und Server agieren dürfen. Das Deadlock-Potenzial, das durch die Änderung entstehen würde, sollte mit der Integration zusätzlicher Deadlock-Detektion-Mechanismen in die Middleware behoben werden.

Außerdem hat es sich gezeigt, dass der bisher mit REQUEST-Nachrichten realisierte synchrone Kommunikationsmechanismus eine Lösung darstellt, bei der Client-Applikationen Anwenderdaten nur mit zusätzlichem Kommunikationsaufwand an den Server übermitteln können. Für eine optimale Ausnutzung sollte die Umsetzung des synchronen Kommunikationsmechanismus so erweitert werden, dass beliebige Anwenderdaten mit den REQUEST-Nachrichten übertragen werden können.

Weiterhin weist der *single threaded* Ansatz des ObjectServer von MiRPA-X ein Konfliktpotenzial bei der Abarbeitung von Konfigurationsnachrichten auf, das im ungünstigsten Fall die Echtzeitfähigkeit der IPC-Nachrichten gefährden kann. Eine Lösung dieses Konflikts bietet die Realisierung von Nebenläufigkeit innerhalb des ObjectServer. Hierbei sollen die Konfigurations- und IPC-Nachrichten durch zwei unterschiedliche Threads mit unterschiedlichen Prioritäten bearbeitet werden.

Schließlich hat sich gezeigt, dass wegen der aktuellen Implementierung des in MiRPA-X integrierten seriellen Token-Scheduling das gesamte auf Mehrkernprozessoren verfügbare Rechenpotenzial nicht genutzt werden kann. Um eine bessere Ausnutzung der Rechenressourcen zu erreichen, müsste die Funktionsweise des Token-Schedulers so geändert werden, dass mehrere Token-Threads, bei Erhalt der Datenintegrität, gleichzeitig aktiviert werden können.

Was die Zuverlässigkeit und Fehlersicherheit im System angeht, hat sich gezeigt, dass die im IAP implementierten Fehlererkennungs- und Fehlerbehandlungsmechanismen für einen sicheren Betrieb unzureichend sind. Hier müssen zusätzliche fehlertolerante Mechanismen eingebaut werden, die bei gelegentlich auftretenden Fehlern den zyklischen Betrieb aufrechterhalten. Außerdem hat sich gezeigt, dass ein Upgrade des Bussystems auf dem IEEE1394 B-Standard die Übertragungszeit der Soll- und Istwerte reduzieren und die Systemperformanz erhöhen würde.

Die Analyse des Hardwaredesigns und der Software-Abläufe auf den Kommunikationsknoten hat gezeigt, dass letztere sowohl in Hard- als auch in Software ein großes Optimierungspotenzial aufweisen. Die aktuelle für den Austausch von Soll- und Istwerten zwischen Steuerungs-PC und den Sensoren und Aktoren anfallende Zeit (ca. 64% eines 1 kHz Zyklus) lässt sich mit einem neuen auf FPGA basierenden Hardwaredesign der Kommunikationsmodule und gezielten Softwareänderungen erheblich (bis unter 10% eines 1 kHz Zyklus) verkürzen.

Nachdem die einzelnen optimierungsfähigen Module und Funktionalitäten in Soft- und Hardware innerhalb der Kommunikations-Infrastruktur identifiziert wurden, wird im nächsten Kapitel zunächst die Umsetzung der Optimierungsmaßnahmen in dem Software-Framework

(MiRPA-X und IAP) beschrieben. Im darauffolgenden Kapitel erfolgt anschließend die Beschreibung des neuen FPGA basierten Hardwaredesigns der Kommunikationsmodule mit entsprechenden softwaretechnischen Optimierungsmaßnahmen.

4 Optimierung des Software-Frameworks

Im letzten Kapitel wurden Funktionalitäten und Module der Kommunikations-Infrastruktur identifiziert, deren Optimierung zur Steigerung der gesamten Systemperformanz beitragen. In diesem Kapitel wird nun beschrieben, wie die einzelnen Optimierungsmaßnahmen des aus MiRPA-X und IAP bestehenden Software-Frameworks umgesetzt wurden.

4.1 Applikationsprozesse mit Client- und Server-Eigenschaften

Wie im Abschnitt 3.1.1 bereits erwähnt, führt eine Änderung des Client/Server-Modells von MiRPA-X zu einer besseren Performanz und einer Vereinfachung des Designs und der Datenflüsse innerhalb der RCA562 Steuerungsarchitektur. In der Tat erlaubt die aktuelle Implementierung von MiRPA-X ausschließlich Client-Applikationen Nachrichten zu initiieren. Zu diesem Zweck ist eine Software-Sperre in den API-Funktionen eingebaut, die während der Nachrichtenregistrierung den Typ der aufrufenden Applikation prüft und die erfolgreiche Funktionsausführung ausschließlich bei Applikationen des Typs CLIENT zulässt.

Damit Server-Applikationen ebenfalls Nachrichten initiieren können, wird lediglich die Software-Sperre aus den entsprechenden MiRPA-X API-Funktionen entfernt. Neben den positiven Auswirkungen auf das Steuerungssystem hat diese strukturelle Änderung den Nachteil, dass sie die inhärente Deadlock-Freiheit der Kommunikationsmechanismen von MiRPA-X aufhebt. Die Kommunikationswege unter MiRPA-X können sich dynamisch ändern. Ein Server A kann eine blockierende Anfrage an einen Server B stellen und der Server B wiederum eine Anfrage an Server A. In diesem Fall führt die Anfrage des Servers B zu einem zyklischen Deadlock. Natürlich kann der Anwender solche einfachen Fälle bereits in der Entwurfsphase vermeiden. Aber oft ist eine Deadlock-Situation in der Praxis nicht einfach zu erkennen. In Abbildung 4-1 beispielsweise wird eine Anfrage vom Server A initiiert und über mehrere Server-Module zum Server E weitergereicht. Schließlich sendet auch der Server E eine Anfrage an den bereits blockierten Server B und verursacht einen Deadlock. Weil sämtliche Anfragen über die Middleware abgewickelt werden, kann letztere die Kommunikationspfade beobachten und Deadlock-Situationen erkennen und aktiv verhindern. Zur Deadlock-Erkennung und Deadlock-Vermeidung wurden zwei Ansätze miteinander verglichen: die Hierarchisierung des Kommunikationsflusses und der Ansatz der Graphentheorie.

Bei der Hierarchisierung des Kommunikationsflusses werden Hierarchieebenen für die Anwenderprozesse von der Middleware in Betracht gezogen. Um einen Deadlock zu verhindern legt MiRPA-X fest, dass blockierende Abfragen ausschließlich von Prozessen einer höheren Hierarchieebene zu Prozessen einer niedrigeren Hierarchieebene (oder umgekehrt) zugelassen werden. Die Hierarchierichtung bleibt dabei von dem Anwender frei wählbar. Bei der Implementierung dieses Ansatzes werden die Prozess-Prioritäten als Hierarchieebenen durch MiRPA-X aufgefasst. Diese Prioritätswerte werden bei der Registrierung der Anwenderprozesse automatisch an MiRPA-X übermittelt. Die Middleware legt eine interne Prioritätstabelle an, die

ein statisches Abbild der Priorität sämtlicher registrierten Anwenderprozesse darstellt. Anhand der Prioritätstabelle kann MiRPA-X im laufenden Betrieb prüfen, ob eine Anfrage die festgelegte Hierarchie des Kommunikationsflusses erfüllt. Falls eine Anfrage die Hierarchie verletzt, dann kann sie zu einem Deadlock führen. In diesem Fall wird sie von MiRPA-X mit entsprechendem Fehlercode zurückgewiesen. Der Vorteil dieses Verfahrens ist die konstante und niedrige verursachte Latenzzeit (< 100 ns). Nachteilig sieht das Verfahren vor, dass Anwenderprozesse eine statische Priorität besitzen, die einmal vom Anwender in der Konfigurationsphase festgelegt wird und zur Laufzeit nicht mehr verändert werden darf. Andersfall würde die MiRPA-X interne Prioritätstabelle nicht mit dem realen Abbild übereinstimmen. Alternativ könnten die Anwenderprozesse über gesonderte Konfigurationsnachrichten die Prioritätstabelle dynamisch aktualisieren. Dies würde aber einen höheren Aufwand seitens des Anwenders erfordern und die Bedienungskomplexität der Middleware steigern. Weiterhin senkt der hierarchische Kommunikationsfluss die Flexibilität des allgemeinen Softwareentwurfs, da die Hälfte der möglichen Kommunikationspfade gesperrt bleibt.

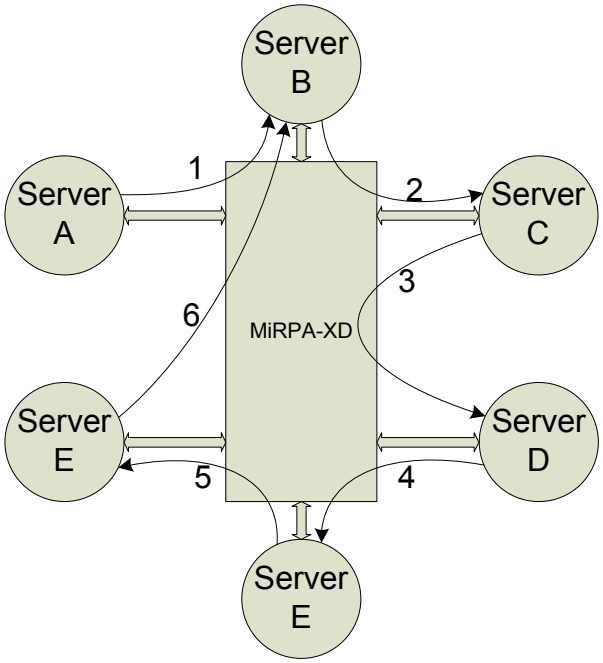

Abbildung 4-1: Zyklischer Deadlock

In dem graphenbasierten Ansatz werden die Abhängigkeiten zwischen den einzelnen Servern als einfacher gerichteter Graph modelliert. In diesem Modell umfasst die Knotenmenge sämtliche in MiRPA-X registrierte Server-Prozesse. Weiterhin steht eine gerichtete Kante zwischen einem Server A und einem Server B dafür, dass Server A eine Anfrage an Server B gestellt hat und blockierend darauf wartet, dass Server B die Anfrage bearbeitet und das Ergebnis zurücksendet. Bevor MiRPA-X eine Anfrage in den Graph aufnimmt, prüft es, ob sie zu einem zyklischen Graph führt. In diesem Fall würde die Ausführung der Anfrage zu einem Deadlock führen. Aus diesem Grund werden solche Anfragen von MiRPA-X mit entsprechendem Fehlercode

zurückgewiesen. In dem in Abbildung 4-1 dargestellten Beispiel würde die Anfrage Nummer 6 von MiRPA-X zurückgewiesen werden, weil sie zu einem zyklischen Graph führen würde. Im Gegensatz zu dem vorherigen Ansatz wird die Flexibilität des Softwareentwurfs nicht beeinträchtigt. Allerdings ist die durch den graphbasierten Ansatz verursachte Latenzzeit abhängig von der Länge des Pfades, der durch aufeinander blockierte Anwenderprozesse im Graph abgebildet wird. Diese Latenzzeit ist allerdings kleiner 1 µs bei einer Pfadlänge bis zu 20 Knoten. Danach steigt der Wert weiter und erreicht 20 µs bei 100 Knoten. Da in der Praxis mit Pfadlängen von 2 bis maximal 10 Knoten gerechnet werden kann, bleibt die Latenzzeit vernachlässigbar. Aufgrund der höheren einhergehenden Flexibilität des Softwareentwurfs wurde der graphenbasierte Ansatz in MiRPA-X realisiert.

4.2 Optimierung der synchronen Kommunikationsvorgänge

Im Abschnitt 3.1.2 wurde bereits gezeigt, dass das Design und die Realisierung der synchronen nachrichtenbasierten Kommunikation (REQUEST) ohne die clientseitige Übermittlung von Applikationsdaten eine suboptimale Lösung bezüglich der erreichbaren Performanz der Kommunikationsvorgänge zwischen Anwenderapplikationen darstellt. Bevor die entsprechende Änderung in der Middleware erläutert wird, muss erklärt werden, wie MiRPA-X die synchrone Kommunikation realisiert.

Bevor Anwenderapplikationen mit Nachrichten kommunizieren können, müssen letztere bei MiRPA-X registriert werden. MiRPA-X stellt zwei API-Funktionen für die Nachrichtenregistrierung zur Verfügung: **MIRPA_CreateMsg()** bei Client-Applikationen und **MIRPA_RegisterMsg()** bei Server-Applikationen**.**

```
unsigned int MIRPA_CreateMsg (const char *pcMsgName,
                              unsigned int uiMsgType,
                              void **ppData,
                              unsigned int uiDataByteSize)

unsigned int MIRPA_RegisterMsg ( const char *pcMsgName,
                                 unsigned int uiMsgType,
                                 void **ppData,
                                 unsigned int uiDataByteSize,
                                 unsigned int **ppuiMsgSpec)
```

Bei der Nachrichtenregistrierung stellt die MiRPA-X API intern zwei Datenpuffer mit spezifischen Parametern für den Nachrichtenversand und den Nachrichtenempfang bereit. **MIRPA_CreateMsg()** und **MIRPA_RegisterMsg()** stellen dem Anwender aber einen einzigen Datenzeiger **ppData** als Parameter zur Verfügung, der der Handhabung von Applikationsdaten dient. Bei Client-Applikationen wird der Datenzeiger intern auf den Speicherbereich innerhalb des Nachrichten-Datenpuffers gesetzt, der für den Nachrichteneingang reserviert ist. Auf diese

Weise arbeitet die Applikation direkt mit einem Datenbereich innerhalb der zu versendenden Nachricht. Bei einem späteren Versenden der Nachricht müssen keine zusätzlichen Kopiervorgänge erfolgen. Dadurch, dass der Datenzeiger mit dem Nachrichteneingangspuffer intern verbunden wird, lässt er sich ausschließlich für den Empfang der Antwort auf REQUEST-Nachrichten verwenden. Daher gibt es für einen Client keine Möglichkeit, mit einer REQUEST-Nachricht umfangreiche Applikationsdaten zu versenden. Auf der Server-Seite sieht die MiRPA-X API vor, dass der Datenzeiger auf den internen Nachrichtenausgangspuffer gesetzt wird.

Damit Applikationsdaten mit REQUEST-Nachrichten versendet werden können, muss ein weiterer Datenzeiger bei der Nachrichtenregistrierung von den API-Funktionen zur Verfügung gestellt werden. In Anlehnung an das von QNX bereitgestellte *Message-Passing* API wurde das entsprechende MiRPA-X API wie folgt erweitert:

```
unsigned int MIRPA_CreateMsg (const char *pcMsgName,
                              void **ppDataTx,
                              unsigned int uiTxDataByteSize,
                              void **ppDataRx,
                              unsigned int uiRxDataByteSize)

unsigned int MIRPA_RegisterMsg (const char *pcMsgName,
                                void **ppDataTx,
                                unsigned int uiTxDataByteSize,
                                void **ppDataRx,
                                unsigned int uiRxDataByteSize,
                                unsigned int **ppuiMsgSpec)
```

Damit können Client-Applikationen Applikationsdaten über den Datenzeiger ppDataTx spezifizieren und mit REQUEST-Nachrichten versenden. Antwort-Daten, die in der ANSWER-Nachricht enthalten sind, werden dagegen über den Datenzeiger ppDataRx empfangen. Der Datenzeiger ppDataTx bzw. ppDataRx wird intern auf den Nachrichtenausgangspuffer bzw. Nachrichteneingangspuffer gesetzt.

Die Erweiterung der API verursacht keine zusätzliche Latenz in der Nachrichtenverwaltung, da der gleiche *Message-Passing* Mechanismus von QNX eingesetzt wird. Durch die API-Erweiterung erübrigt sich der zur Übermittlung von Applikationsdaten bis dahin übliche doppelte Aufruf einer COMMAND-Nachricht, gefolgt von einer REQUEST-Nachricht.

4.3 Nebenläufigkeit in der Nachrichtenverwaltung

Das Konfliktpotenzial, das der *single threaded* Ansatz von MiRPA-X bei der Bearbeitung von Konfiguration und IPC-Nachrichten birgt, wurde bereits im Abschnitt 3.1.3 erwähnt. Um dieses

Problem zu lösen, wird eine *multi threaded* Erweiterung des Middleware-Kerns vorgenommen. Dabei wird ein zweiter Kommunikationskanal für die Bearbeitung von Konfigurationsnachrichten zur Verfügung gestellt. Auf diese Weise ist die Middleware in der Lage, IPC- und Konfigurationsnachrichten „quasi“ parallel in zwei nebenläufigen Threads zu bearbeiten.

Bei der Umsetzung dieser funktionalen Erweiterung werden sämtliche Nachrichten weiterhin von dem „ObjectServer“-Thread empfangen, der mit hoher Priorität ausgeführt wird. Falls es sich um eine Konfigurationsnachricht handelt, leitet der ObjectServer die Nachricht zur Bearbeitung an den zweiten, mit niedriger Priorität laufenden Konfigurations-Thread (siehe „Config“ in Abbildung 4-2). Aufgrund des Prioritätsunterschieds ist gewährleistet, dass der Eingang einer IPC-Nachricht immer die Bearbeitung von Konfigurationsnachrichten unterbricht und die IPC-Nachrichten ihre Echtzeiteigenschaften in jedem Fall beibehalten. Da Konfigurationsnachrichten den Inhalt der internen Nachrichtenverwaltungsdaten von MiRPA-X verändern, kann es durch die Unterbrechung des Konfigurations-Threads prinzipiell zu einem Konflikt mit dem IPC-Thread und Dateninkonsistenz beim Zugriff auf gemeinsam genutzte Verwaltungsdaten kommen.

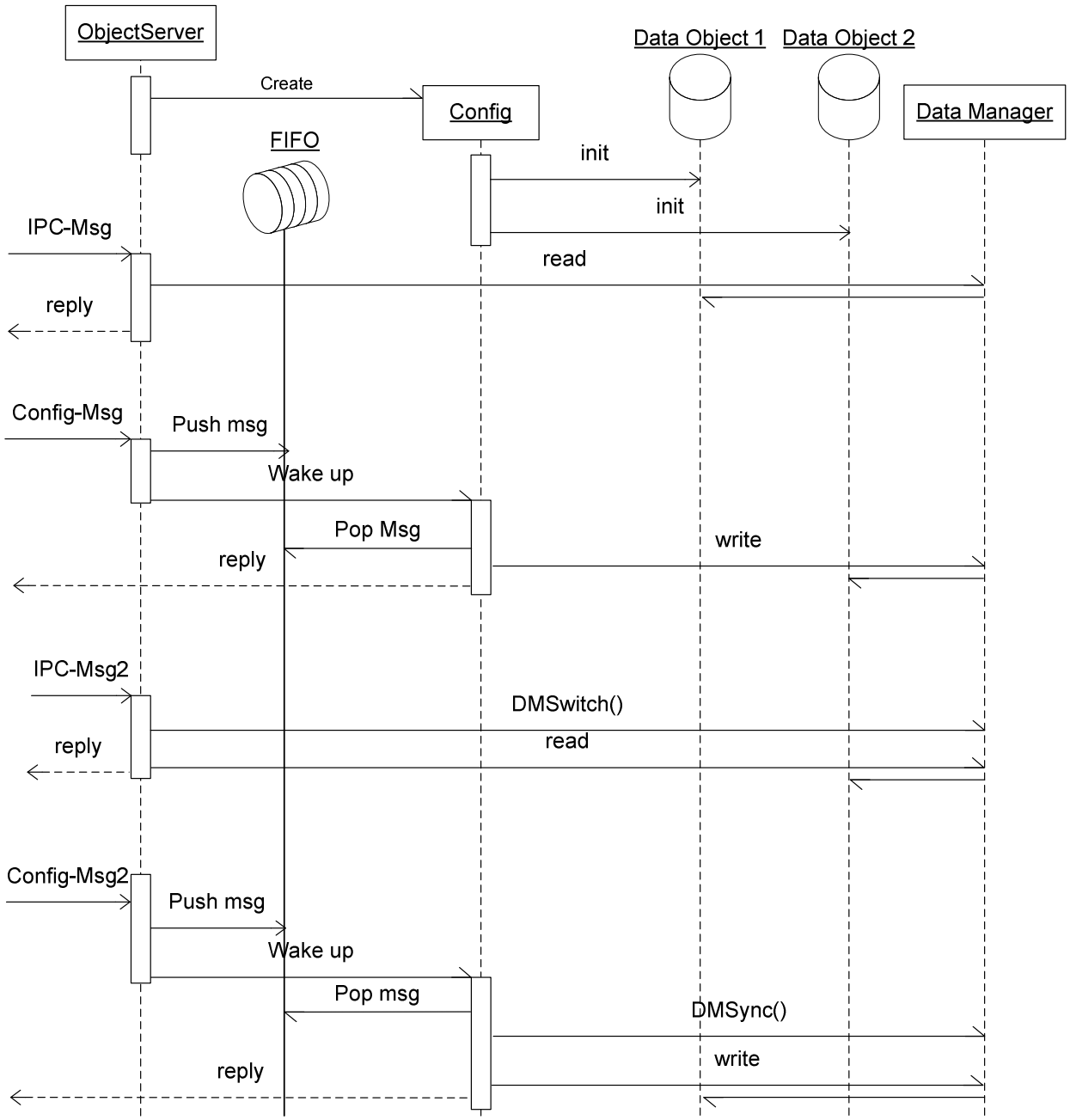

Abbildung 4-2: Multi threaded Ansatz zur Nachrichtenverwaltung in MiRPA-X, Thread-Synchronisation basierend auf doppelt ausgeführten Verwaltungsdaten.

Weil der IPC-Thread in dem Echtzeitkontext unmittelbar nach Eingang einer IPC-Nachricht ausgeführt werden muss, können die Zugriffskonflikte hier nicht mit einem einfachen Synchronisationsmechanismus (z.B. Mutex) gelöst werden. In Abbildung 4-2 ist die erarbeitete und sehr effiziente Lösung für das Synchronisationsproblem dargestellt. Sie basiert auf einer doppelten Ausführung der Verwaltungsdaten in zwei separaten Datenobjekten. Die nebenläufigen Threads greifen nie auf dasselbe Datenobjekt zu. Der Konfigurations-Thread verändert den Inhalt des Datenobjekts, während der IPC-Thread bei der Nachrichtenzustellung

nur lesend darauf zugreift. Der Zugriff auf die Datenobjekte erfolgt über einen Zugriffsmanager, der ebenso die Synchronisierung der Datenbestände durchführt.

In Abbildung 4-2 wird anhand eines Sequenzdiagramms erläutert, wie die Synchronisierung abläuft. Die Datenobjekte werden dabei als DatenObject1 und DatenObject2 dargestellt. Bei der Initialisierung des ObjectServer wird ein zusätzlicher Konfigurations-Thread (Config) erzeugt. Der Konfigurations-Thread initialisiert daraufhin die Datenobjekte DatenObject1 und DatenObject2. Das Sequenzdiagramm zeigt exemplarisch die Abarbeitung einer wechselnden Folge von Konfigurations- und IPC-Nachrichten. Der ObjectServer-Thread fungiert dabei als IPC-Thread. Bei der Zustellung der ersten IPC-Nachricht greift der IPC-Thread lesend auf DatenObject1 zu. Die erste Konfigurationsnachricht „Config-Msg“ wird ebenfalls vom ObjectServer-Thread empfangen, deren Abarbeitung wird aber an den Thread „Config“ delegiert. Dafür macht der ObjectServer einen entsprechenden Eintrag in das FIFO (Push msg) und weckt anschließend den Konfigurations-Thread. Bei der Registrierung der Ressource greift der Konfigurations-Thread auf DataObject2 zu und verändert dessen Inhalt. Danach sind die Verwaltungsdaten in DataObject2 aktuell, während DataObject1 veraltete Daten beinhaltet. Die Registrierung wird mit einer Reply-Nachricht abgeschlossen. Nach dem Eingang der zweiten IPC-Nachricht „IPC-Msg2“, bewirkt der Zugriffsmanager ein internes Umschalten (DMSwitch) der Objektreferenzen. Das Umschalten bewirkt, dass der IPC-Thread auf das zuvor aktualisierte DataObject2 zugreift und so die zuletzt vorgenommene Konfiguration berücksichtigt, während der Konfigurations-Thread sich auf das veraltete DataObject1 bezieht. Bei der nächsten Konfigurationsnachricht „Config-Msg2“ wird der Inhalt der zwei Datenobjekte miteinander abgeglichen (DMSync). Hier sei angemerkt, dass die relativ zeitintensive Synchronisierung der Datenbestände durch den Konfigurations-Thread bei geringerer Priorität ausgeführt wird.

Nun stellt sich natürlich die Frage, wie dieser Ansatz die Verzögerung der Abarbeitung der IPC-Nachrichten durch die Konfigurationsnachrichten aufheben kann. Dies wird in Abbildung 4-3 (markierter Bereich) veranschaulicht. Dort wird exemplarisch gezeigt, wie der Eingang einer IPC-Nachricht die Abarbeitung einer Konfigurationsnachricht unterbricht. Bevor der Konfigurations-Thread die Registrierung der Ressource abschließt und DataObject2 aktualisiert, wird die IPC-Nachricht „IPC-Msg“ empfangen. Aufgrund der höheren Priorität unterbricht der IPC-Thread die Ausführung des „Config“-Threads. Um die mögliche Dateninkonsistenz im DataObject2 zu vermeiden, greift er bei der Nachrichtenzustellung auf die Verwaltungsdaten in DataObject1 zurück.

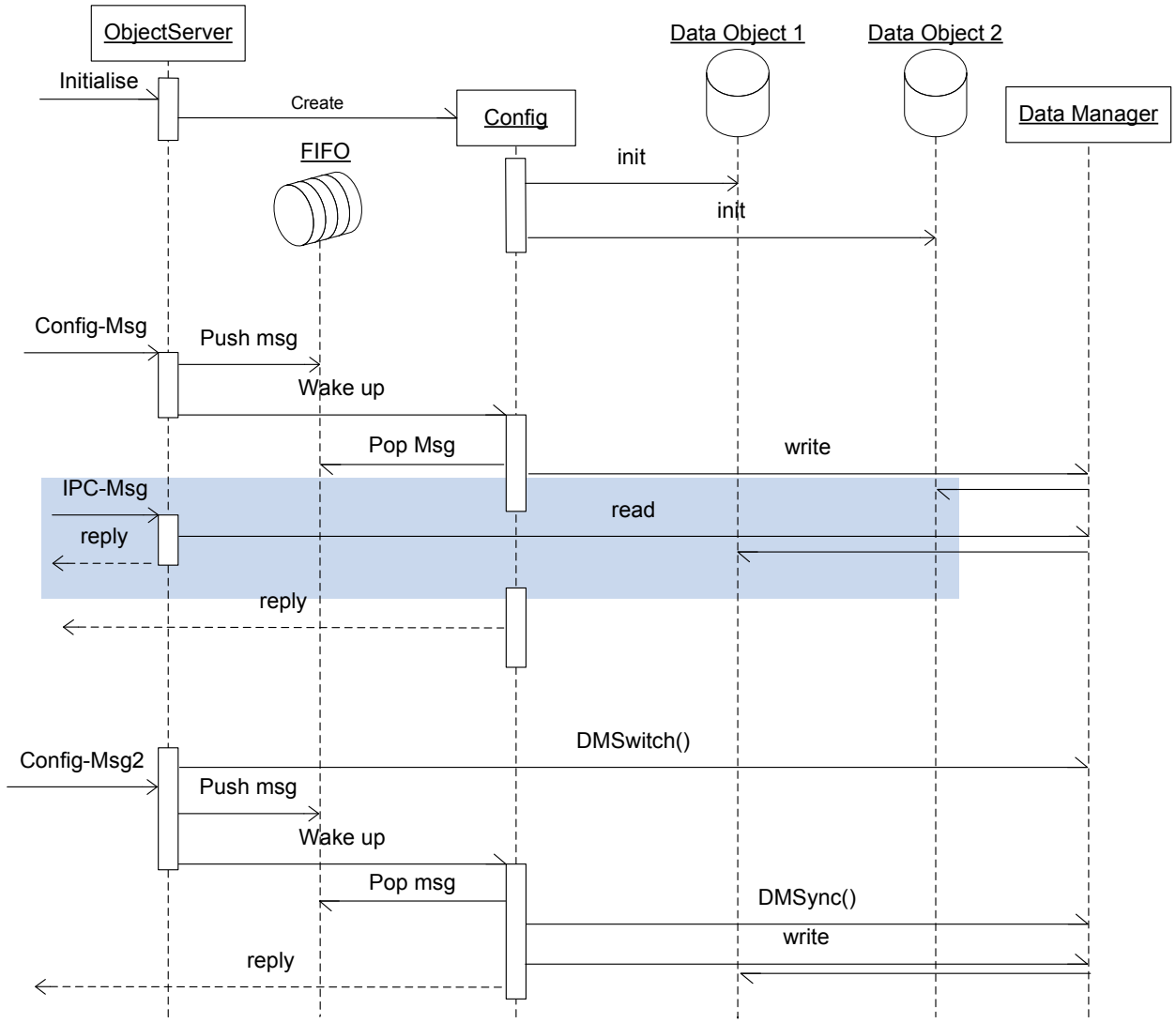

Abbildung 4-3: Multi threaded Ansatz zur Nachrichtenverwaltung in MiRPA-X - Unterbrechung des „Config“-Thread wegen priorisierter Ausführung einer IPC-Nachricht.

Steuerung des Zugriffs auf die Datenobjekte

Der Konfigurations-Thread und der IPC-Thread greifen nicht direkt auf die Datenobjekte zu. Der Zugriff wird stattdessen durch den Datenmanager gesteuert. Der Datenmanager stellt eine Schnittstelle zur Verfügung, die aus zwei Zeigerreferenzen besteht: a) einem aktiven Zeiger, der stets auf das aktuelle Datenobjekt zeigt und b) einem passiven Zeiger, der auf das veraltete Datenobjekt zeigt. Der Zugriff des IPC-Threads auf die Verwaltungsdaten erfolgt stets über den aktiven Zeiger. Der Konfigurations-Thread dagegen greift auf die Verwaltungsdaten über den passiven Zeiger zu. Die Aktivitäten des Datenmanagers entsprechen einem Automaten mit vier Zuständen, wie in Abbildung 4-4 a dargestellt. Während der Initialisierung werden die Datenobjekte erstellt und deren Inhalte miteinander abgeglichen. Anschließend wechselt der Manager in den IDLE-Zustand. In diesem Zustand bewirkt der Eingang einer IPC-Nachricht

keinen Zustandswechsel. Erst der Eingang einer Konfigurationsnachricht bewirkt einen Wechsel in den Zustand CHANGED; doch vor dem Zustandswechsel wird die entsprechende Ressource-Konfiguration durchgeführt und das passive Datenobjekt aktualisiert (siehe Abbildung 4-4 b).

Nach Eintreffen einer IPC-Nachricht werden die internen Zeigerreferenzen so umgeschaltet, dass der aktive Zeiger (IPC in Abbildung 4-4 b) auf das zuvor aktualisierte Datenobjekt und der passive Zeiger (Config in Abbildung 4-4 b) auf das bis jetzt veraltete Datenobjekt zeigt. Anschließend wechselt der Datenmanager in den Zustand SWITCHED_IPC. Nach Eingang der nächsten Konfigurationsnachricht werden erst die Datenobjekte miteinander synchronisiert, anschließend wird das passive Datenobjekt aktualisiert und wieder in den Zustand CHANGED gewechselt. Im CHANGED-Zustand bewirkt das Eintreffen einer Konfigurationsnachricht keinen Zustandswechsel. Es werden lediglich die Datenobjekte miteinander synchronisiert (iii), die Zeigerreferenzen intern umgeschaltet (ii) und letztlich die Ressource-Konfiguration ausgeführt und das passive Datenobjekt aktualisiert (i).

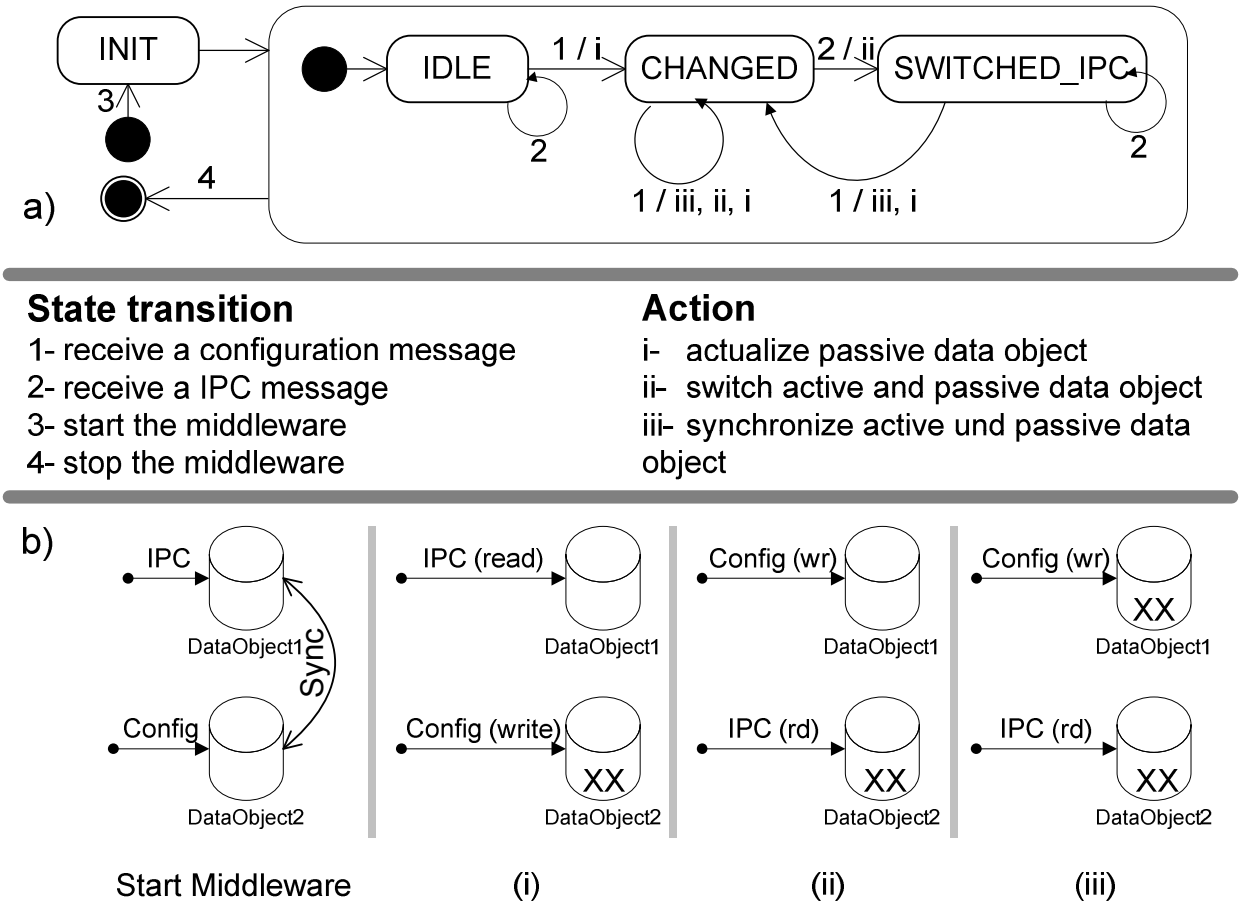

Abbildung 4-4: Aktivitäten des Datenmanagers: a) Zustandsdiagramm des Managers, b) Steuerung des Zugriffs auf die Datenobjekte

4.4 Erweiterung des Token-Schedulings für Mehrkernprozessoren

Die Notwendigkeit einer Erweiterung der Scheduler-Funktionalität, um eine optimale Nutzung der auf Mehrkernprozessoren verfügbaren Rechenleistung zu erzielen, wurde bereits im Abschnitt 3.1.4 erwähnt. Die entsprechenden Funktionsänderungen und Erweiterungen, die in [DMM08] veröffentlicht wurden, lassen sich anhand des im Abbildung 4-5 dargestellten Beispiels erläutern.

Abbildung 4-5 a) entspricht dem bisherigen seriellen Scheduling von sechs Token-Threads (T1 bis T6) auf einem Einkernprozessor. Die Token-Threads werden nacheinander vom Scheduler via Token-Versand aktiviert. Der Token-Versand wird hierbei mittels blockierendem synchronem *Message-Passing* realisiert. Aufgrund der blockierenden Eigenschaft kann der Scheduler nicht mehrere Token hintereinander versenden. Demzufolge ist eine gleichzeitige Aktivierung mehrerer Token-Threads nicht möglich.

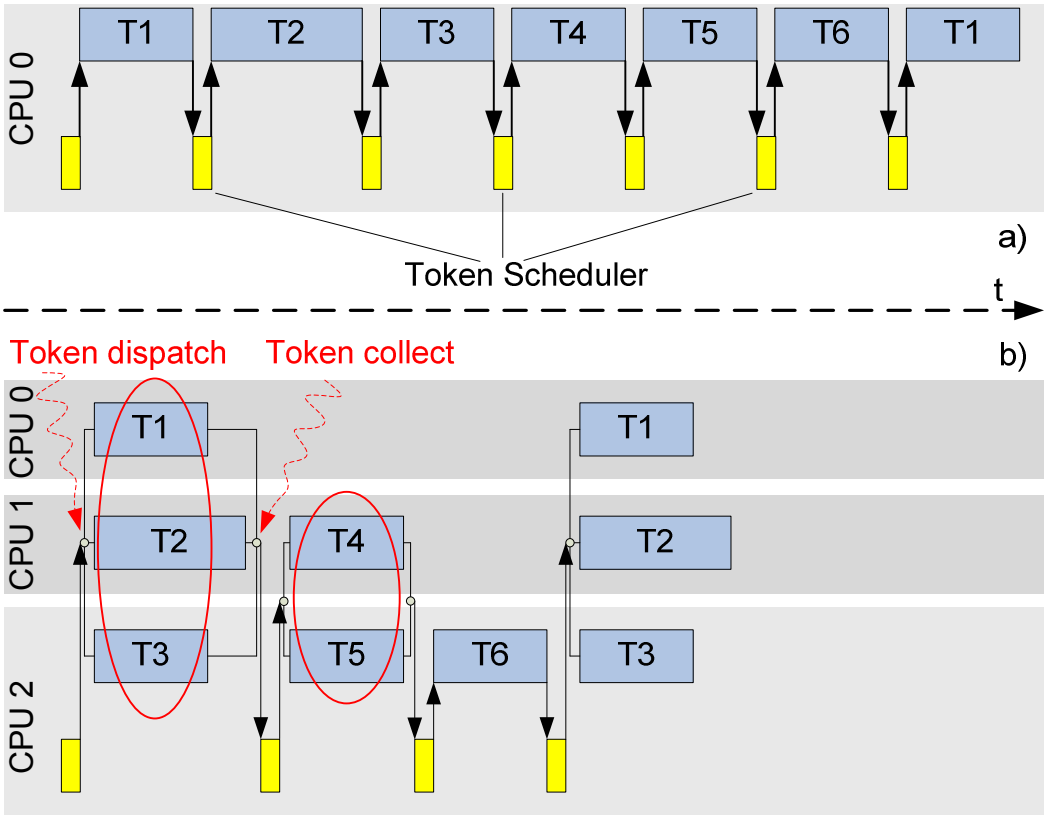

Abbildung 4-5: Erweiterung des Token-Scheduling für den Einsatz auf Mehrkernprozessoren.

Auf einer Mehrkernprozessor-Plattform setzt die gleichzeitige Aktivierung mehrerer Token-Threads einen nicht blockierenden Mechanismus für den Token-Versand voraus. Aus diesem Grund wurde der Token-Versand neu entworfen und mit der Übermittlung von nicht-

blockierenden *Pulse*-Nachrichten realisiert. Die funktionale Erweiterung des Token-Scheduling für den Einsatz auf Mehrkernprozessoren ist in Abbildung 4-5 b) dargestellt. Um die verfügbare Rechenleistung besser zu nutzen, werden Token-Threads, die keine Datenabhängigkeiten untereinander aufweisen, in Modulen zusammengefasst. Zur Veranschaulichung sind solche Module beispielsweise in Abbildung 4-5 b) eingekreist. Die Token-Threads eines jeden Moduls werden anschließend zur parallelen Ausführung auf die vorhandenen CPUs verteilt. In diesem Kontext aktiviert der Scheduler jedes Modul, indem er entsprechend soviel Token sendet, wie Token-Threads in dem jeweiligen Modul enthalten sind. In Abbildung 4-5 b) sendet der Scheduler entsprechend drei Token an die Token-Threads T1, T2 und T3. Die nebenläufigen Token-Threads, die unterschiedliche Ausführungszeiten haben, senden den Token nach abgeschlossener Ausführung an den Scheduler zurück. Um die Synchronisation mit den nächsten Modulen zu erhalten und die Datenintegrität zu gewährleisten, werden Token-Sammelpunkte (Token Collect) eingesetzt. An einem Sammelpunkt wartet der Scheduler solange blockiert, bis er sämtliche zuvor versendeten Token zurückerhält, bevor er das nächste Modul aktiviert.

Konfiguration des Scheduler

Bevor die erweiterte Scheduler-Funktionalität angewendet werden kann, muss sie vorher konfiguriert werden. Bei der Konfiguration der Scheduler muss der Anwender angeben:

- welche Token-Threads als Module zusammengefasst und parallel ausgeführt werden sollen,
- welcher Token-Thread auf welchem CPU ausgeführt werden soll,
- in welcher Reihenfolge die Token-Threads aktiviert werden sollen.

Die statischen Konfigurationsdaten werden in eine Konfigurationsdatei gespeichert. Die in Tabelle 1 enthaltenen Daten entsprechen der Scheduler-Konfiguration für das in Abbildung 4-5 b) dargestelle Beispiel. In der Zeile „Token-Name“ werden sämtliche Token-Threads aufgelistet, die innerhalb des Token-Zyklus ausgeführt werden. Die Reihenfolge der Eingaben entspricht der Reihenfolge, in der die Token-Threads aktiviert werden sollen. Der Nebenläufigkeitsindex legt fest, welche Token-Threads parallel auf verschiedenen CPUs ausgeführt werden sollen. Ein Indexwert gleich 0 besagt, dass der entsprechende Token-Thread nicht parallel mit anderen Token-Threads ausgeführt werden darf. Indexwerte ungleich 0 deutet hingegen auf eine parallele Ausführung hin. Sämtliche Token-Threads mit dem gleichen Indexwert (ausgenommen Indexwert 0) sollen parallel ausgeführt werden. In diesem Fall müssen die Token-Threads unterschiedlicher CPUs zur Ausführung zugeordnet werden. Die Zuordnung erfolgt anhand der CPU-Maske. Entsprechend der Scheduler-Konfiguration in Tabelle 1 sind zwei Module definiert, die jeweils die Token-Threads T1, T2, T3 und T4, T5 enthalten. T1, T2 und T3 sollen jeweils auf CPU 0, 1 und 2 ausgeführt werden.

Tabelle 1: exemplarische Konfiguration des Scheduler für die parallele Ausführung auf einem Multikernprozessor.

Token-Name	T1	T2	T3	T4	T5	T6
Nebenläufigkeitsindex	1	1	1	2	2	0
CPU-Maske	0	1	2	1	2	2

Ergebniss

Dadurch, dass Token-Threads auf Mehrkernprozessoren parallel ausgeführt werden, können kleinere Token-Zyklen realisiert und folglich die gesamte Performanz des Steuerungssystems erhöht werden. Die Durchlaufzeit eines Token-Zyklus kann theoretisch bis um den Faktor n reduziert werden (n gleich Anzahl der verfügbaren CPUs). Auf einem Dual-Core-System wurde eine Verbesserung der Durchlaufzeit von 45% für den Token-Zyklus gemessen. Weiterhin kann der Scheduler die parallele Ausführung von Token-Threads auf ein System mit bis zu 32 Prozessorkernen handhaben. Dabei beträgt die Latenz des Token-Versands 6 µs.

4.5 IAP

Um das IAP im Bezug auf fehlende DDTs (*Missing DDT*) fehlertolerant zu manchen, muss deren Fehlerbehandlung geändert werden. Das Blockdiagramm im Abbildung 4-6 beschreibt die neue tolerante Fehlerbehandlung im IAP-Protokoll. Am Anfang eines jeden Zyklus prüft das IAP nach, ob sämtliche erwartete DDTs eingetroffen sind. Falls mindestens ein DDT fehlt, dann wird der Fehler als *Missing DDT* erkannt.

Bisher hat das IAP auf ein *Missing DDT* mit einem Wechsel in den STOP-Zustand und einem Abbruch der zyklischen Kommunikation reagiert. Um die Fehlertoleranz in das IAP zu integrieren, wurde es so geändert, dass es nicht mehr unmittelbar nach dem ersten Missing DDT in den STOP-Zustand wechselt, sondern die höhere Applikationsebene (Steuerungssoftware) benachrichtigt und erst bei wiederholten Missing DDTs innerhalb kürzerer Zeit in den Stop-Zustand wechselt. Nach einem *Missing DDT* erhöht das IAP einen internen Fehlerzähler um 500, der nach dem Systemstart mit dem Wert 0 initialisiert wurde. Anschließend wird die Steuerungssoftware über einen Shared-Memory-Bereich benachrichtigt, falls der Fehlerzähler einen Wert zwischen 1 und 999 hat. Ist der Wert größer als 1000, wechselt das IAP in den STOP-Zustand und bricht die zyklische Kommunikation ab. Der Fehlerzähler wird in jedem Zyklus um 1 heruntergezählt, falls kein Missing DDT auftritt.

Nachdem die Steuerung den Fehler erfasst hat, kann sie spätestens im nächsten Zyklus darauf reagieren. Sie kann beispielweise neue Sollwerte für eine kontrollierte Bremsung der Antriebe generieren und über die noch bestehende Kommunikation an die Antriebe schicken.

Durch die Integration eines fehlertoleranten Mechanismus in die Fehlerbehandlung des IAP-Protokolls wird gewährleistet, dass die Kommunikation robuster gegenüber gelegentlich auftretenden Telegrammfehlern wird.

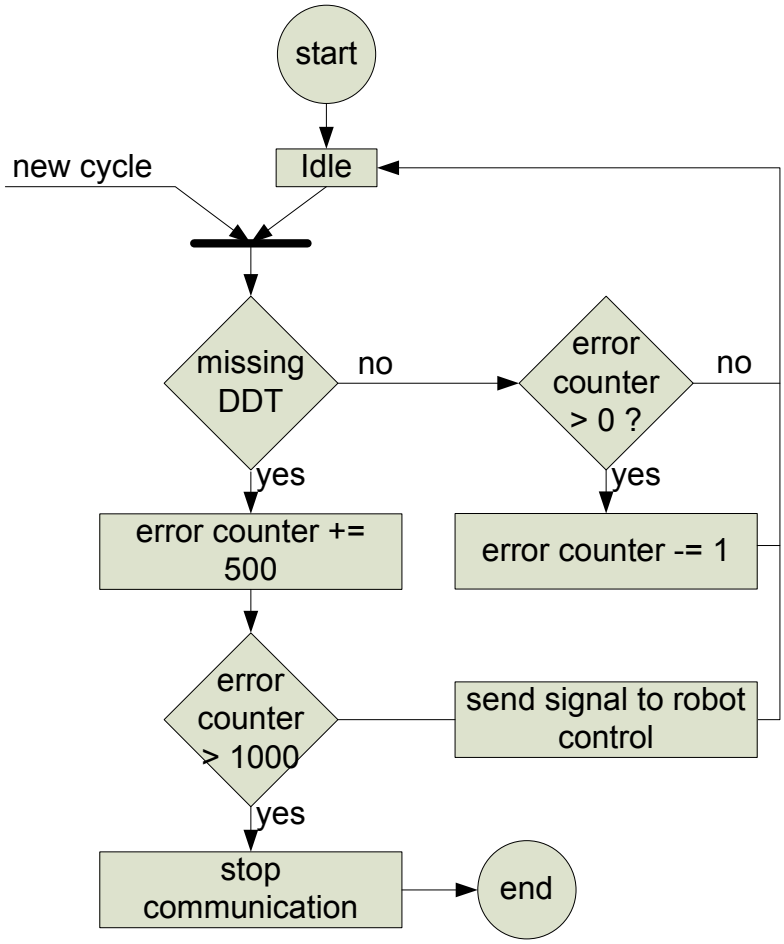

Abbildung 4-6: Erweiterte Fehlerbehandlung im IAP-Protokoll

4.6 Zusammenfassung

In diesem Abschnitt wurde die Implementierung der im Kapitel 3 aufgestellten Optimierungsmaßnahmen für das Software-Framework der Kommunikations-Infrastruktur beschrieben.

Insbesondere wurde das Client/Server Applikationsmodell von MiRAP-X so geändert, dass Server-Applikationen ebenfalls Nachrichten initiieren können. Dafür wurde lediglich eine Software-Sperre aus der für den Nachrichtenversand zuständigen API-Funktion entfernt. Zusätzlich wurde ein graphenbasierter Mechanismus zur aktiven Deadlock-Erkennung und Vermeidung in MiRPA-X integriert.

Außerdem wurde die Struktur der synchronen REQUEST-Nachrichten so geändert, dass Applikationsdaten nun mit einer REQUEST-Anfrage übermittelt werden können und gleichzeitig Kommunikationsbandbreite gespart wird. In der Praxis konnte diese Änderung durch eine Erweiterung der für den Nachrichtenversand zuständigen API-Funktion um einen zusätzlichen ausgehenden Datenpuffer realisiert werden.

Um die auf Mehrkernprozessoren vorhandenen Rechenressourcen besser zu nutzen, wurde die Funktionsweise des Scheduler so erweitert, dass er die Aktivierung und die parallele Ausführung mehrerer Token-Threads unterstützt. Mit der Erweiterung lässt sich theoretisch das volle Rechenpotenzial von Mehrkernprozessoren ausnutzen.

Das bei der Bearbeitung von Konfiguration und IPC-Nachrichten wegen der *single threaded* Auslegung des ObjectServer von MiRPA-X vorhandene Konfliktpotenzial wurde durch eine Erweiterung des Middleware-Kerns um einen zusätzlichen Thread beseitigt, dem die Verwaltung von Konfigurationsnachrichten zugeordnet wurde. Die Konflikte zwischen beiden Threads, die beim Zugriff auf interne Verwaltungsdaten auftreten können, wurden mit einem sehr effizienten Synchronisationsansatz gelöst, der die doppelte Ausführung der Verwaltungsdaten in zwei separaten Datenobjekten basiert.

Schließlich konnte durch die Integration eines fehlertoleranten Mechanismus in die Fehlerbehandlung des IAP-Protokolls erreicht werden, dass die zyklische Kommunikation robuster gegenüber gelegentlich auftretenden Telegrammfehlern wurde.

Zusammenfassend tragen die in diesem Kapitel beschrieben Änderungen an der Middleware und den IAP-Funktionalitäten dazu bei, dass der Entwicklungsaufwand für Anwendersoftware reduziert, eine höhere Performanz der einzelnen Kommunikationsmechanismen (bis um 45 % beim erweiterten Token-Scheduling für Mehrkernproessoren) erzielt und ein robusteres und fehlertolerantes System aufgebaut werden können. Im nächsten Kapitel werden die Optimierungen beschrieben, die an den Kommunikationsknoten durchgeführt wurden.

5 Optimierung der Kommunikationsmodule

Dieses Kapitel behandelt die Optimierung der Kommunikationsmodule. Die Kommunikationsmodule bieten die größte Optimierungsmöglichkeit hinsichtlich der Performanz des gesamten Systems. Um eine schnelle Datenverarbeitung zu erzielen, ist es notwendig, ein neues Design sowohl in Soft- als auch in Hardware zu erstellen. Als Zielplattform wurde ein FPGA ausgewählt; dieser Ansatz ermöglicht es, verschiedene Systemkomponenten auf einem einzigen Chip zu integrieren. Neben der hohen Performanz bedeutet dies vor allem einen geringeren Platzbedarf. Durch die Systemintegration lässt sich der Stapelaufbau der bisherigen Kommunikationsmodule (Abbildung 2-6) durch eine Einplatinenlösung ersetzen. Der bisherige Stapelaufbau konnte die geometrische Spezifikation für den Einbau in die vorgesehenen Einsteckplätze des Frequenzumrichter-Hardware-Moduls (FHM) nicht einhalten. Deshalb wurden zusätzliche Verlängerungsplatinen eingesetzt, um die Kommunikationsmodule mit dem Frequenzumrichter zu verbinden. Allerdings weist dieser Umweg eine mechanische Instabilität und eine erhöhte EMV-Störanfälligkeit auf, die von der hohen Anzahl der Steckkontakte und dem Einsatz der Verlängerungsplatine herrühren.

In Folgenden werden die Optimierungen beschrieben, die durch ein neues Hard- und Software-Design realisiert wurden. Anschließend werden die Auswirkungen der Optimierungen auf die gesamte Systemperformanz dargestellt.

5.1 Hardware-Design

Abbildung 5-1 stellt das FPGA-basierte Hardware-Design schematisch dar. Als Plattform wurde der für eingebettete Systeme optimierte Virtex-4 FX12 Chip von Xilinx ausgewählt. Der Virtex-4 FX 12 zeichnet sich durch einen geringen Leistungsverbrauch aus. Er bietet einen eingebetteten PowerPC-Prozessor, der mit bis zu 450 MHz betrieben werden und dabei eine Leistung von mehr als 700 Dhrystone[12] MIPS liefern kann. Die Systementwicklung wurde durch die XPS (Xilinx Platform Studio) Softwarekette von Xilinx unterstützt.

Im Gegensatz zu dem bisherigen Design passen nun alle elektronischen Komponenten auf eine einzelne Platine. Neben den Funktionalitäten des IEEE1394 LLC Moduls und des DRPAM-Moduls wird eine Prozessoreinheit, mit schnellem internem Cache zur effizienten Datenverarbeitung, in das FPGA integriert. Der IEEE1394 PHY sowie andere Komponenten (SRAM, PROM, Flash, UART) werden hingegen als diskrete IC-Bausteine eingebaut.

[12] Benchmark zum Vergleich der Leistungsfähigkeit unterschiedlicher Rechner oder Compiler. Dhrystone MIPS: Million Dhrystone-Instruktionen pro Sekunde

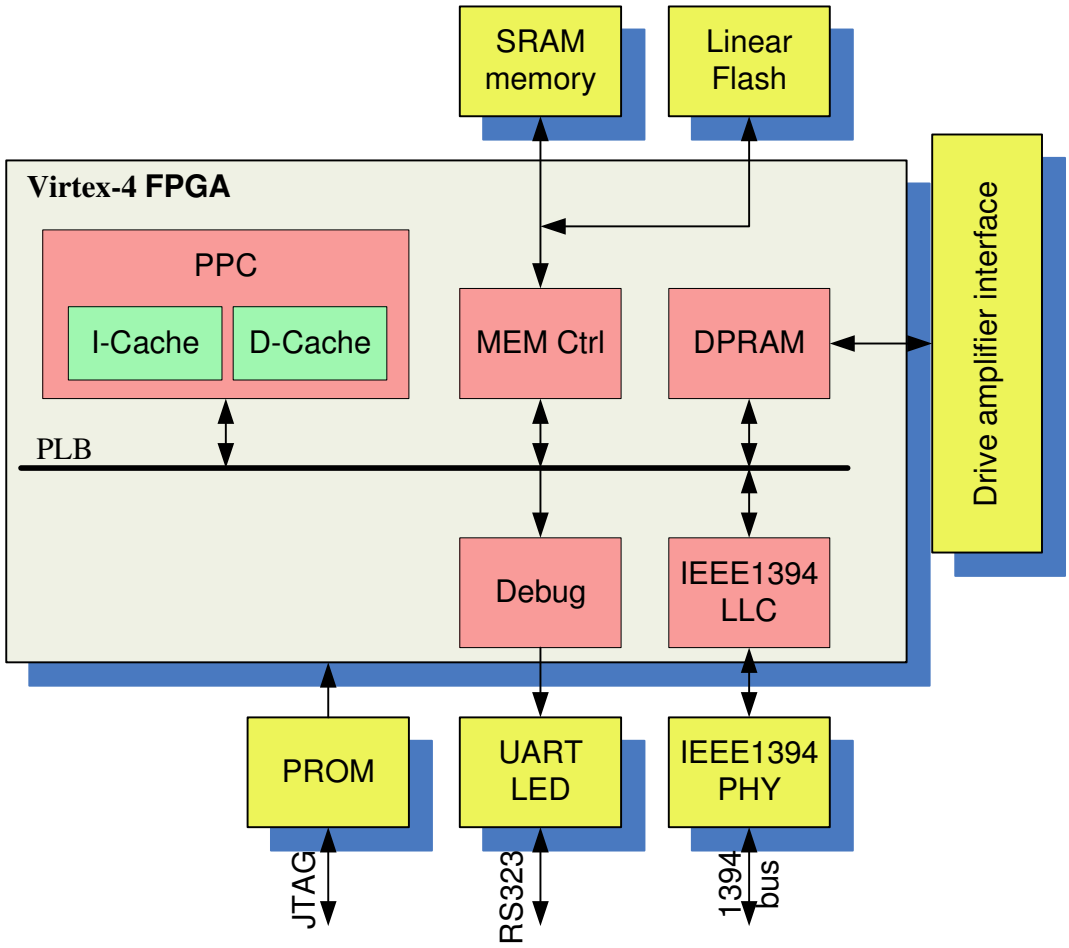

Abbildung 5-1: FPGA-basiertes Hardware-Design eines Kommunikationsmoduls

5.1.1 LLC Core

Der 1394b Link Layer Core [Cor07] namens FireLink LLC Core der Firma „DAP Technology“ [Dap09] ist kompatibel zu der IEEE 1394b – 2002™ Spezifikation [IEE08] und unterstützt Übertragungsgeschwindigkeiten bis zu 800 Mbps. Ein Blockdiagramm des FireLink LLC Core ist in Abbildung 5-2 dargestellt. Der Paketdatenfluss ist prozessorgesteuert; das bedeutet, jedes empfangene/zu sendende Paket muss aus einem Dual Ported RAM (DPRAM) Buffer gelesen bzw. in ein DPRAM Buffer geschrieben werden. Das DPRAM ist in dem FireLink LLC Core enthalten und hat eine variable Größe, die standardmäßig 4 KBytes beträgt. Diese Größe kann aber der maximalen Paketgröße angepasst werden. Der FireLink Core bietet ein generisches synchrones Host-Interface für den Zugriff auf den DPRAM-Datenbuffer. Der Zugriff auf die Kontroll- und Statusregister erfolgt über ein separates synchrones Interface. Außerdem sind isochrone Ports im FireLink Core enthalten. Das Ziel eines isochronen Ports ist es, dedizierte Hardware (z.B. Bild/Video versendende/empfangende Hardware) zu unterstützen, die die Verarbeitung von Datenströmen autonom ohne den Einsatz des Prozessors erledigen. Über diese Ports lassen sich die isochronen Pakete mit einer hohen Effizienz verarbeiten.

In dem FPGA-Design ersetzt der FireLink Core die bisher auf dem DSP-board verwendete Standard Link Layer Controller IC-Komponente TSB42AB4 von TI[13]. Der FireLink Core lässt sich sowohl über den OPB[14]-Interface als auch über den PLB[15]-Interface mit dem Host verbinden. Auf der Seite des FireWire- Busses bietet der Core ein Interface zu dem IEEE1394b PHY-Chip. Die auf DPRAM basierenden Sende- und Empfangseinheiten haben gegenüber der FIFO-basierten Struktur des TSB42AB4-IC den Vorteil, dass bei der Verarbeitung von MDTs die Daten nicht zwischengespeichert werden müssen. Die Daten können stattdessen direkt über das Host-Interface selektiv geschrieben und gelesen werden. Folgerichtig verkürzt sich die Datenverarbeitungszeit innerhalb der Kommunikationsmodule allein durch die Integration des FireLink Core. Darüber hinaus wird die Datenverarbeitungszeit wegen des Ausfalls von Kopieroperationen unabhängig von der Größe des empfangenen Telegramms.

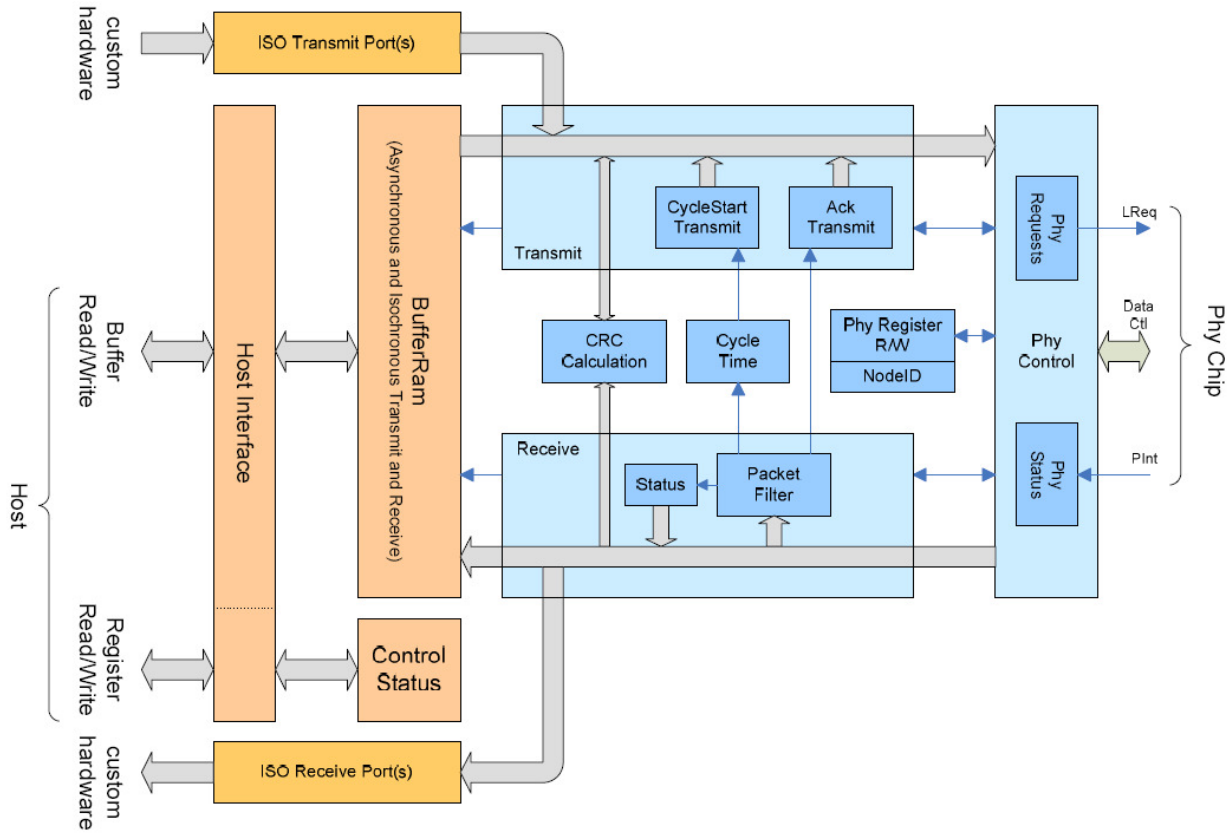

Abbildung 5-2: Blockdiagramm des FireLink 1394b Link Layer Controller Core[16]

[13] Texas Instruments

[14] On-chip Peripheral Bus

[15] Processor Local Bus

[16] Quelle [Cor07]

5.1.2 DPRAM Core

Das DPRAM ist die physische Plattform, auf der die Steuerungsdaten mit dem Frequenzumrichter ausgetauscht werden. Die Integration des DPRAM IP[17]-Core in das Design hat den Vorteil, dass schnellere Datenzugriffe seitens des µC realisiert werden können. Ein DPRAM-IP-Core lässt sich mit Hilfe der Xilinx-Werkzeuge einfach generieren. Ein solcher IP-Core besteht aus einem synchronen Block-RAM mit zwei Standard-Schnittstellen. Über die erste Schnittstelle wird das Block-RAM an den PLB-Bus angeschlossen; auf diese Weise kommt ebenso die Verbindung mit dem µC zustande. Über den PLB-Bus lassen sich außerdem 1-Byte, 2-Bytes und 4-Bytes Datenzugriffe realisieren. Dies hat positive Auswirkungen auf die Ausführungszeit des Austauschs von Sensor/Aktor-Daten mit dem Frequenzumrichter. Die zweite Schnittstelle dient dem Anschluss des Frequenzumrichters. Die ausgewählte Speichergröße des IP-Core von 2 KBytes bietet ausreichend Platz, um einen einwandfreien Datenaustausch mit dem Frequenzumrichter zu realisieren.

In Gegensatz zu dem bisher eingesetzten DPRAM-IC [CYP97] stellt der generierte DPRAM-Core ein Interface zur Anbindung des Frequenzumrichters zur Verfügung, das aber der Spezifikation des Frequenzumrichters nicht vollständig entspricht. Aus diesem Grund muss dieses Interface angepasst werden.

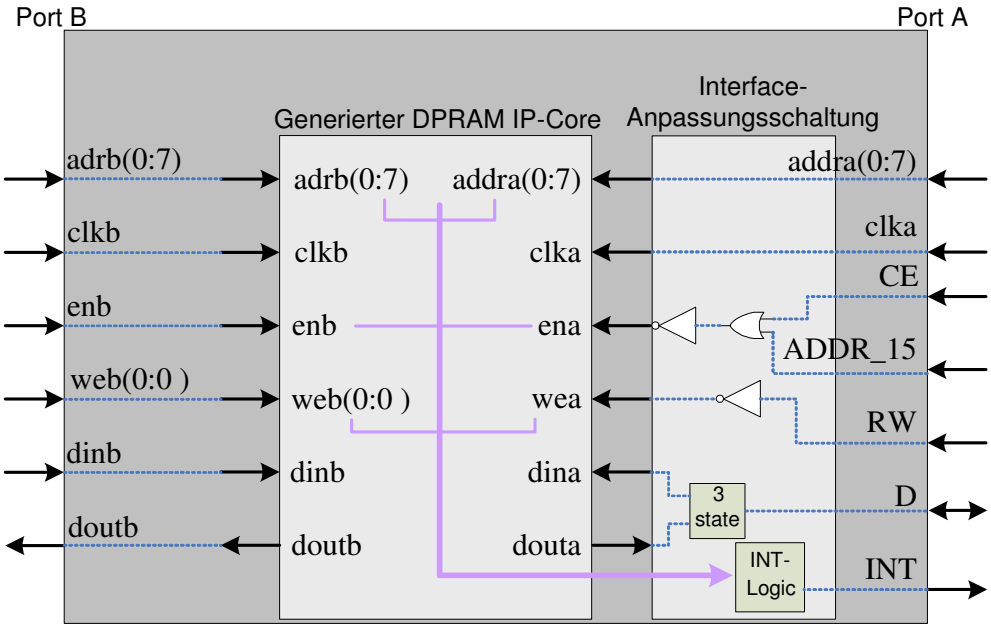

Abbildung 5-3: Schematische Darstellung eines DPRAM-IP-Cores mit angepasstem Interface zum Frequenzumrichter

[17] IP: Intellectual Property

Abbildung 5-3 zeigt den schematischen Aufbau des angepassten DPRAM-IP-Cores. Über eine Anpassungsschaltung wurde das entsprechende Interface zum Frequenzumrichter nachgebildet. Die Adressleitungen „addra(0:7)“ und der Clock „clka“ wurden ohne Änderungen durch die Anpassungsschaltung durchgeschleift und an die entsprechenden Eingänge des generierten DPRAM-IP-Core angeschlossen. Das Signal „Read/Write“ (WR) wurde negiert und ebenso an den entsprechenden Eingang des generierten Cores angeschlossen. Die Signale „Chip Enable“ (CE) und „ADDR_15“ wurden über ein logisches ODER-Gatter verbunden und das resultierende Signal anschließend negiert und an den „Enable“ Port des generierten Cores angeschlossen. Durch diese Verschaltung werden unerwünschte Zugriffe seitens des Frequenzumrichters ausgeschlossen. Dies ist deshalb möglich, weil das aus der Adressleitung des Frequenzumrichters entnommene Signal „ADDR15“ festlegt, ob der Frequenzumrichter auf den Core zugreift. Der Datenausgang „douta“ und der Dateneingang „dina“ des generierten Cores wurden über eine Tri-State-Schaltung zusammengefasst und an die bidirektionale Datenleitung „D“ angeschlossen. Außerdem wurde eine Interrupt-Leitung „INT“ zum Interface hinzugefügt. Eine Interrupt-Leitung ist deshalb erforderlich, weil sie das Medium darstellt, worüber Synchronisationsimpulse generiert und an den Frequenzumrichter übertragen werden. Für das Setzten und Zurücksetzen des Interrupts werden zwei Speicheradressen im DPRAM-IP-Core vereinbart: INT_SET_ADDR und INT_RESET_ADDR. Das Interrupt wird gesetzt, wenn der µC schreibend auf die Speicheradresse INT_SET_ADDR zugreift. Zurückgesetzt wird er, wenn der Frequenzumrichter schreibend auf die Speicheradresse INT_RESET_ADDR zugreift.

5.1.3 Microcontroller

Ein Microcontroller lässt sich in das FPGA-Design entweder als Soft- oder als Hard-Core integrieren. Soft-Cores liegen als Quellcode oder in Form einer Netzliste vor und werden im frei programmierbaren Bereich des FPGAs implementiert. Ein typisches Beispiel für einen Microcontroller Soft-Core ist der MicroBlaze [Xil08], welcher mit einer Taktfrequenz bis zu 100 MHz konfiguriert und samt seinen Programmen bei Bedarf in ein FPGA integriert werden kann. Hard-Cores hingegen sind als fertige Schaltungen herstellerseitig unveränderbar in den Chip des FPGAs integriert. Der Vorteil dabei ist, dass Hard-Cores weniger Chipfläche belegen und meist auch schneller als mit frei programmierter Logik implementierte Soft-Cores arbeiten können. Nachteilig ist die fehlende Möglichkeit, eigene Adaptionen anzubringen oder eine Portierung zu anderen Logikfamilien, die nicht über die meist sehr spezifischen Hard-Cores verfügen, durchzuführen. Im ausgewählten „Xilinx Virtex 4 FX12“ FPGA ist der leistungsfähige Prozessor PowerPC 405 [Xil03] als Hard-Core bereits enthalten. Der PowerPC kann mit einer Taktfrequenz bis zu 300 MHz konfiguriert werden.

Aufgrund des bereits in der Entwurfsphase festgelegten Ziels, die Performanz der Datenverarbeitung innerhalb der Kommunikationsmodule zu erhöhen, wurde der leistungsfähigere PowerPC Hard-Core dem MicroBlaze Soft-Core vorgezogen. Spätere Performanzmessungen haben die Entscheidung für den PowerPC Hard-Core bestätigt. Bei einem

Betrieb mit aktiviertem Cache konnten gegenüber dem Microblaze Soft-Core Performanzerhöhungen bis zum Faktor 8 verzeichnet werden.

5.1.4 Weitere Systemkomponenten

Zur Realisierung weiterer Funktionalitäten werden folgende weitere Komponenten in das Design als diskrete Bausteine integriert:

PROM: Zur dauerhaften Speicherung der generierten FPGA-Konfiguration ist ein von Xilinx speziell entwickelter 1 MB großer, wieder programmierbarer PROM-Baustein [Xil99] vorgesehen. Beim Einschalten wird das FPGA automatisch mit der im PROM gespeicherten Konfigurationsdatei geladen.

Linear Flash: Ein 2 MB großer Flash-Baustein dient der dauerhaften Speicherung der Applikationssoftware. Letztere wird mittels eines Bootloaders beim Einschalten in den Arbeitsspeicher (SRAM) kopiert.

SRAM: Der im Virtex 4 FX12 FPGA zur Verfügung stehende Arbeitspeicher wird mit einem 1 MB großen SRAM-Baustein erweitert. Alternativ kann ein größeres FPGA mit entsprechend größerem internem RAM eingesetzt werden. Aus Kostengründen wurde diese Möglichkeit aber nicht umgesetzt.

FireWire PHY: Zur Anbindung an die FireWire physikalische Schicht ist der Einsatz eines PHY-Chip erforderlich. Hier fiel die Wahl auf den Chip „TSB81BA3D“ von Texas Instruments. Der PHY-Chip wird über das vom LLC-Core zur Verfügung gestellte Interface angeschlossen.

Pegelanpassung an Frequenzumrichter: Die Kommunikation mit dem Freuquenzumrichter über das Dual Port RAM erfordert eine Pegelanpassung. Die 5V-Pegel des Antriebsverstärkers müssen auf die IO-Pegel des FPGAs (3,3V) umgesetzt werden. Hierzu werden einfache Bustreiber eingesetzt.

Debug-Modul: Um die Applikationssoftware testen und Performanzmessungen durchführen zu können, wurde ein Debug-Modul in das Design integriert. Mit Hilfe des Debug-Moduls werden ausgewählte Signale aus dem FPGA nach außen geführt und zur Darstellung von Systemverhalten an LEDs und Anschlussstifte angeschlossen. Weiterhin bietet das Debug-Modul eine serielle Schnittstelle, über die Softwarezustände und Debug-Ausgaben in Textform übermittelt werden können.

DSP-Adapter: das FPGA-Design ist in der derzeitigen Ausführung nur für Kommunikationsmodule ausgelegt, die mit dem Frequenzumrichter Daten im zyklischen Betrieb austauschen. Um die Vorteile des neuen Designs auf andere DSP-basierte Module (z.B. Digital-IO-Modul, Analog-Modul) übertragen zu können, ist eine DSP-Anpassungsschaltung im Design

vorgesehen. Hierfür ist aber die Nachbildung eines EMIF (*External Memory Interface*) in Form eines IP-Cores notwendig.

5.1.5 Prototypische Realisierung

Abbildung 5-4 zeigt einen Prototyp des FPGA-basierten Kommunikationsmoduls. Durch den kompakten Aufbau (Vergleich mit Abbildung 2-6) konnte die geometrische Spezifikation erfüllt werden, die für Erweiterungskarten des Frequenzumrichter-Hardware-Moduls (FHM) gelten. Dadurch konnten FPGA-Karten, im Gegensatz zu den bisherigen DSP-Karten, vollständig in die FHM-Gehäuse eingesteckt und gleichzeitig vor den starken elektromagnetischen Feldern abgeschirmt werden. Außerdem konnte durch den kompakten Aufbau erreicht werden, dass zusätzliche potentielle EMV-Störquellen eliminiert wurden. Solche potentielle Störquellen stellten die bisherigen Erweiterungs-Platinen zum Anschluss der früheren DSP-basierten Kommunikationsmodule an den FHM dar.

Der IEEE1394 Standard spezifiziert keinen Steckverbinder mit industriellem Standard. Um dennoch die Betriebsicherheit zu erhöhen und unerwünschtes Lösen der elektrischen Verbindungen zu unterbinden, wurde eine eigens entworfene Metall-Vorrichtung zur Befestigung der IEEE1394 Stecker an die FPGA-Platine angebracht. Weiterhin weist die Platine einen geringen Energieverbrauch auf: die Leistung beträgt 1,9 W bei 24 V Versorgungsspannung.

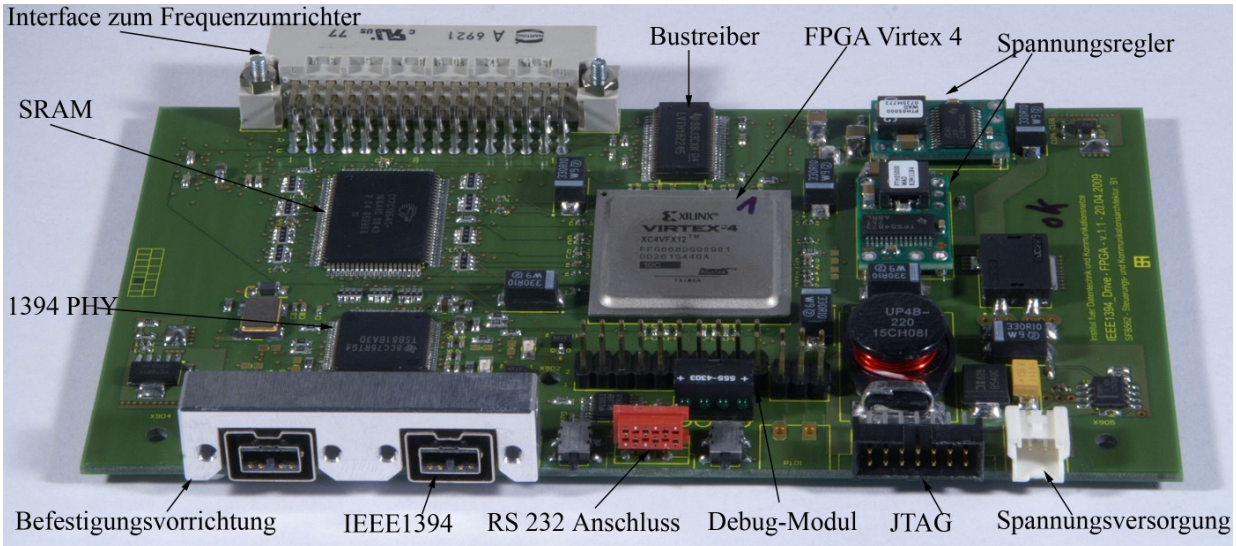

Abbildung 5-4: Prototyp des FPGA-basierten Kommunikationsmoduls

5.2 Software-Optimierungen

Eine Performanzerhöhung des Kommunikationsmoduls lässt sich nicht nur durch Hardware-Änderungen erreichen, sondern auch durch eine damit einhergehende Möglichkeit zur Optimierung des Software-Verhaltens. In Abbildung 5-5 und Abbildung 5-6 werden die Timing-Diagramme der alten (DSP) und der neuen optimierten (FPGA) Software gegenübergestellt. Verglichen werden Software-Abläufe, die zwischen MDT-Empfang und DDT-Versand stattfinden. Als Vergleichskriterium gilt der zeitliche Abstand zwischen MDT und DDT auf dem IEEE1394 Bus. Während dieser Abstand mit der alten Software noch ca. 480 µs beträgt (Abbildung 5-5), ist er mit der neuen Software kleiner als 15 µs (Abbildung 5-6). Dies entspricht einem Verbesserungsfaktor von 32.

Die Software-Änderungen, die diese Verbesserung herbeiführen, sind unterschiedlich. Erstens lässt sich das MDT durch die Integration des FireLink Cores schneller abarbeiten. In der Tat entfallen zeitaufwendige Kopie-Operationen der MDT-Nutzdaten aus dem LLC FIFO zum Hauptspeicher, da der FireLink Core direkte Speicherzugriffe auf interne Sende- und Empfangspuffer zulässt. Die Verarbeitungszeit des MDTs lässt sich damit von ca. 150 µs auf 8 µs reduzieren.

Zweitens wird die Generierung des Synchronisationsimpulses für die Frequenzumrichter an den Empfang des MDTs gekoppelt. Der Synchronisationsimpuls wird dementsprechend während der MDT-Abarbeitung generiert, und zwar bevor die Sollwerte in das DPRAM aktualisiert werden. Unmittelbar nach Ablauf der MDT-Abarbeitung wird das DDT aus den aktuell im DPRAM vorliegenden Istwerten zusammengestellt und über IEEE1394 zum Steuerungsrechner zurück geschickt. Die Kopplung des Synchronisationsimpulses mit dem MDT hat den Vorteil, dass sämtliche Timer-Events, die eingebauten Sicherheitszeiten (55µs) sowie der Aufwand zur Synchronisation der Systemuhren entfallen. Auf diese Weise werden insgesamt mehr als 80 µsec eingespart.

Drittens wird eine Änderung des Protokolls für den Datenaustausch mit dem Frequenzumrichter vorgenommen. Innerhalb eines 1 kHz Kommunikationszyklus führt der Frequenzumrichter 4-mal einen internen zyklischen Task jeweils mit einem Abstand von 250 µs aus. Der Task, der unmittelbar nach dem Erhalt des Synchronisationsimpulses zur Ausführung kommt, nimmt den Datenaustausch vor. Mit dem Synchronisationsimpuls als Referenz werden nach 120 µs Sollwerte aus dem DPRAM gelesen und nach 180 µs aktuelle Istwerte in das DPRAM geschrieben (Abbildung 5-5). Dennoch, um DDTs unmittelbar nach der Bearbeitung des MDTs senden zu können, müssen die Istwerte bereits vor dem MDT Empfang im DPRAM vorliegen. Um dies zu erreichen, sieht die Protokolländerung vor, dass der Frequenzumrichter die Ausführung des Datenaustauschs in zwei Tasks aufsplittert. Demzufolge erfolgt die Istwert-Aktualisierung in der Task unmittelbar vor dem Synchronisationsimpuls, während das Auslesen der Sollwerte weiterhin nach dem Synchronisationsimpuls erfolgt (Abbildung 5-6). Auf Anfrage wurde eine entsprechende Firmware-Version des Frequenzumrichters vom Hersteller realisiert.

Steuerungstechnisch stellt diese Änderung auf den ersten Blick einen Nachteil dar, weil die Istwerte sich nicht direkt auf die aktuellen Sollwerte beziehen. Dennoch lassen sich durch die Wegoptimierung der 250 µs Totzeit höhere Steuerungszyklen und insgesamt eine bessere Regelgüte erreichen.

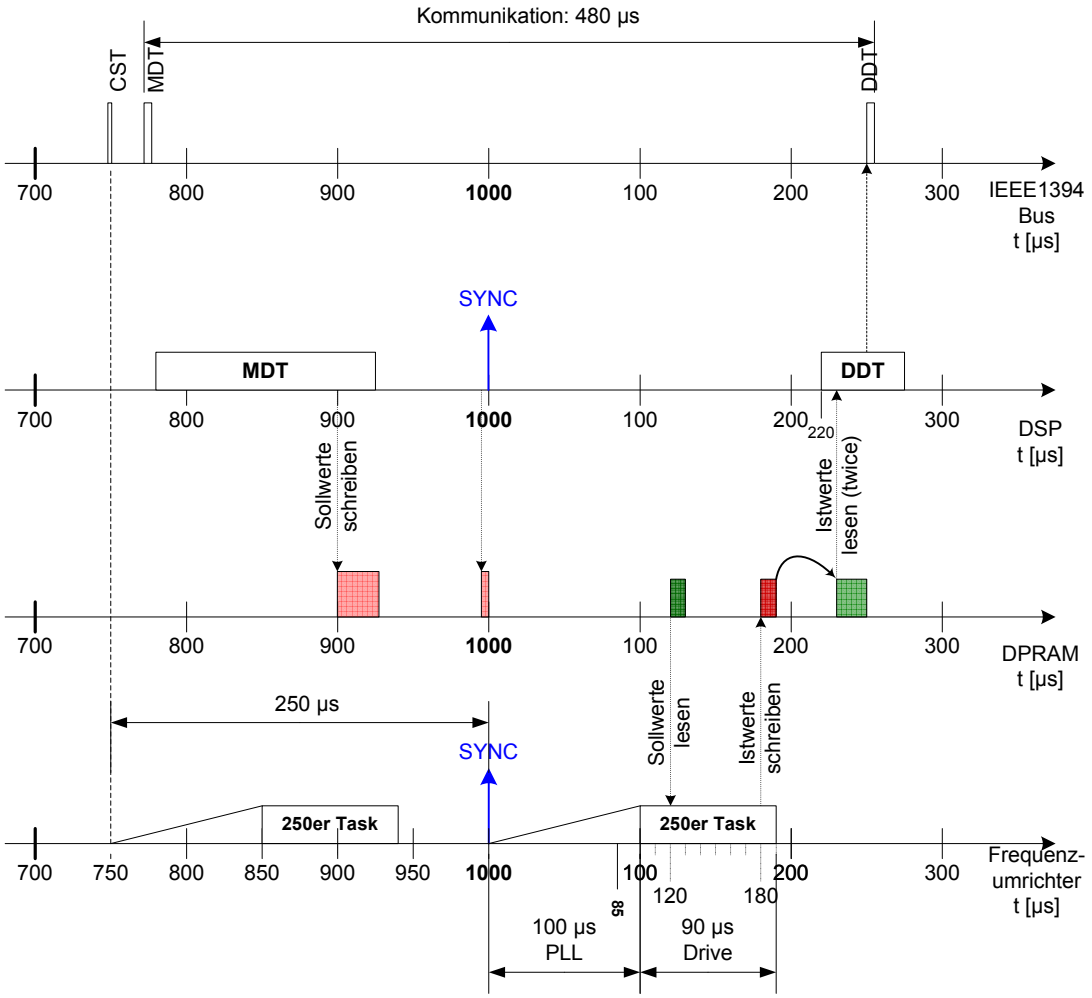

Abbildung 5-5: Datenverarbeitung: altes Timing-Diagramm (DSP)

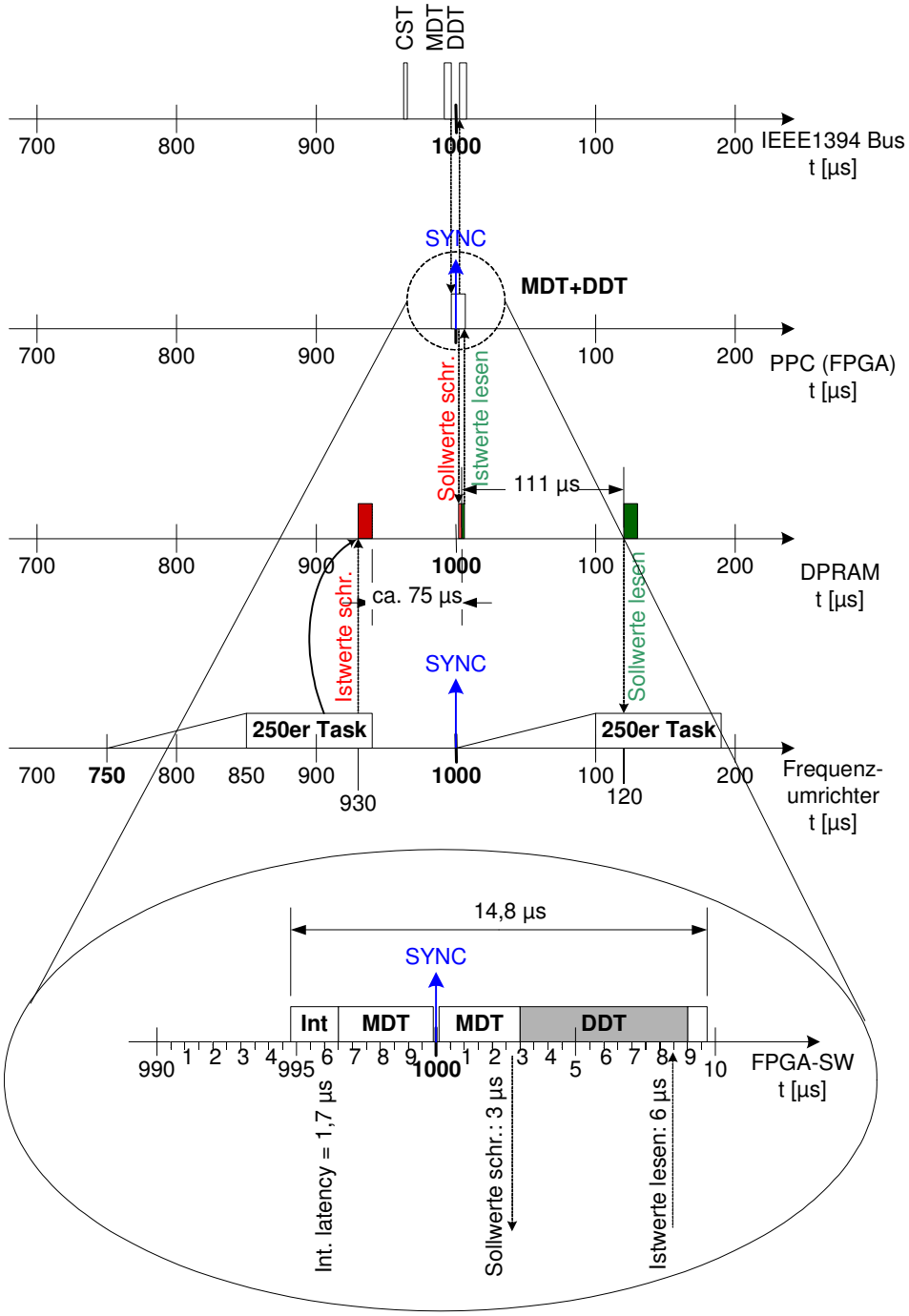

Abbildung 5-6: Datenverarbeitung: neues Timing-Diagramm (PPC)

5.3 Performanz des Kommunikationssystems nach Optimierung

Um die Auswirkungen der Optimierungsmaßnahmen auf die Performanz des gesamten Systems besser in Zahlen zu fassen, werden folgende Mess- und Vergleichsindizes festgelegt: der maximal realisierbare Steuerungszyklus, der prozentuale Anteil der Kommunikationslatenz an dem jeweiligen Steuerungszyklus und die auf dem Steuerungsrechner zur Verfügung stehende

Applikationszeit. Die Kommunikationslatenz umfasst die Latenz des IAP-Protokolls. Um die Systemperformanz zu ermitteln, wird dementsprechend ein IAP-Zyklus berücksichtigt. Die Systemkonfiguration sieht einen Steuerungsrechner und 6 Kommunikationsmodule vor. Gemessen wird vom Eingang eines CSTs im Steuerungsrechner, über dem Versand eines MDT und dem Empfang von 6 DDTs, bis zum nächsten CST.

Die Messergebnisse, bei Nutzung des IEEE1394 A-Standards, sind in Abbildung 5-11 dargestellt. Die Kommunikationszeit beträgt 60 µs und ist unabhängig von dem eingestellten Zyklus. Im Vergleich zum ursprünglichen System (Abbildung 3-5 in Abschnitt 3.4) entspricht dieser Wert einer Verbesserung um den Faktor 10. Bei einer Zyklusfrequenz von 1 kHz nimmt die Kommunikation 6% der Zykluszeit in Anspruch, während 94% (940 µs) der Steuerungsapplikation zur Verfügung stehen. Bei 2 kHz kommt der prozentuale Anteil der Kommunikation auf 12%, während 88% der Zykluszeit der Steuerungsapplikation zur Verfügung stehen. Bei 250 µs Zykluszeit bleiben immer noch 190 µs für Steuerungsapplikationen. Da die Kommunikationszeit 60 µs beträgt, ist sogar eine Zyklusfrequenz von 8 kHz realisierbar. Allerdings dürfte die dabei zur Verfügung stehende Applikationszeit von 65 µs nur den Leistungsanforderungen von Steuerungsapplikationen mit geringer Komplexität und CPU-Last genügen.

5.4 Auswirkung der Optimierung auf die Regelgüte

Um die Auswirkung der Optimierungen auf die Regelgüte zu zeigen, wurden entsprechende Messungen am TRIGLIDE-Roboter durchgeführt. Dazu wurde der Manipulator entlang der X-Achse mit einer maximalen Geschwindigkeit von jeweils 0,4 m/s und 4,6 m/s hin und her bewegt. Als Gütemaß wurde der Regelfehler (Differenz zwischen der Soll- und Ist-Position) festgelegt. Die Messung wurde sowohl bei Regelfrequenzen von 1 und 2 kHz durchgeführt. Die Messergebnisse sind zum Vergleich in Abbildung 5-7, Abbildung 5-8, Abbildung 5-9 und Abbildung 5-10 dargestellt.

In Abbildung 5-7 wird der Regelfehler über die Zeit bei einer Regelfrequenz von 1 kHz und einer maximalen Geschwindigkeit von 0,4 m/s dargestellt. Der wiederkehrende Vorzeichenwechsel in der Graphik lässt sich auf die Hin- und Her-Bewegung auf der X-Achse zurückführen. Der Betrag des Regelfehlers pendelt um 0,4 mm. Mit der Verdoppelung der Regelfrequenz (Abbildung 5-8) halbiert sich der Betrag des Regelfehlers und pendelt um 0,2 mm. Dieser Effekt zeigt sich auch bei einer erhöhten Gechwindigkeit von 4,6 m/s (Abbildung 5-9 und Abbildung 5-10). Dies zeigt, dass die Erhöhung der Regelfrequenz, die durch die in dieser Arbeit durchgeführten Optimierungen möglich wurde, eine direkte messbare und positive Auswirkung auf die Regelgüte hat.

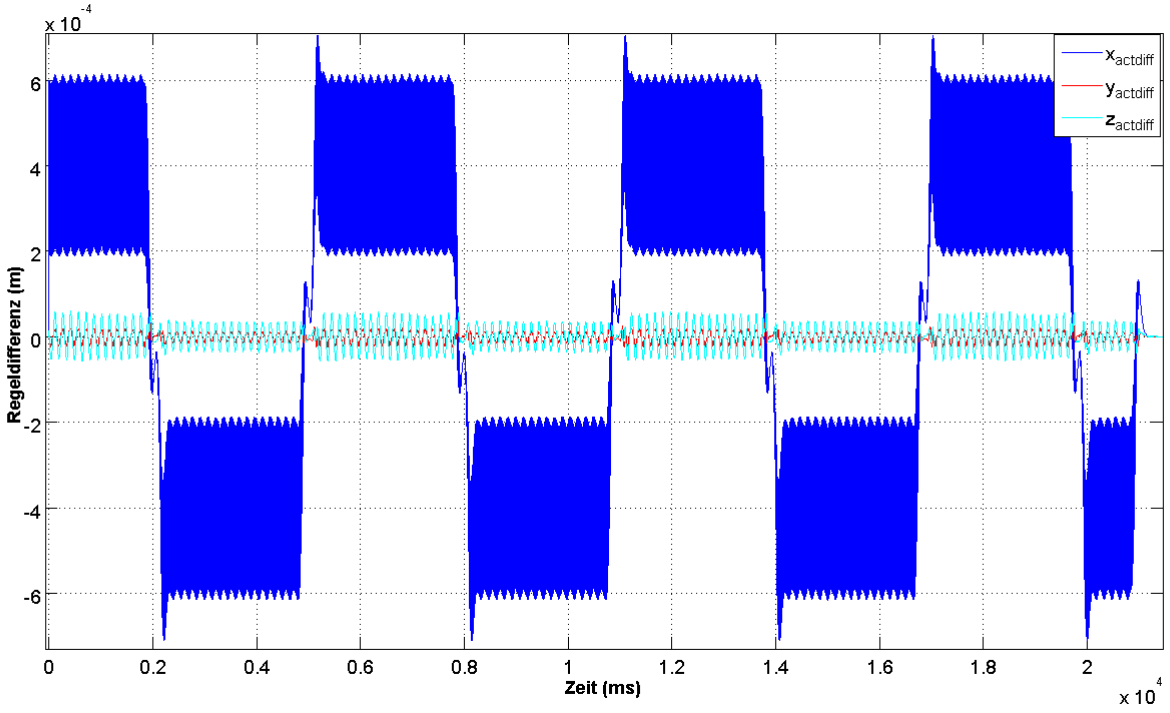

Abbildung 5-7: Regelfehler in der X-Richtung bei 1000 Hz Regelungsfrequenz und 0.4 m/s Geschwindigkeit

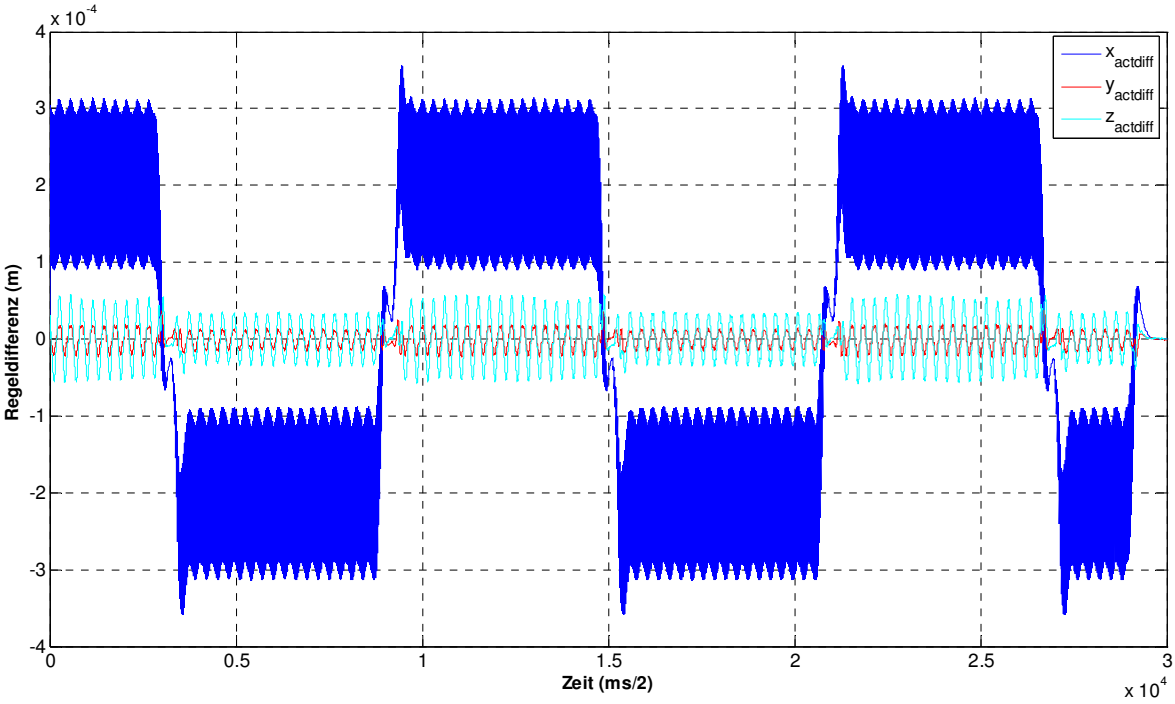

Abbildung 5-8: Regelfehler in der X-Richtung bei 2000 Hz Regelungsfrequenz 0.4 m/s Geschwindigkeit

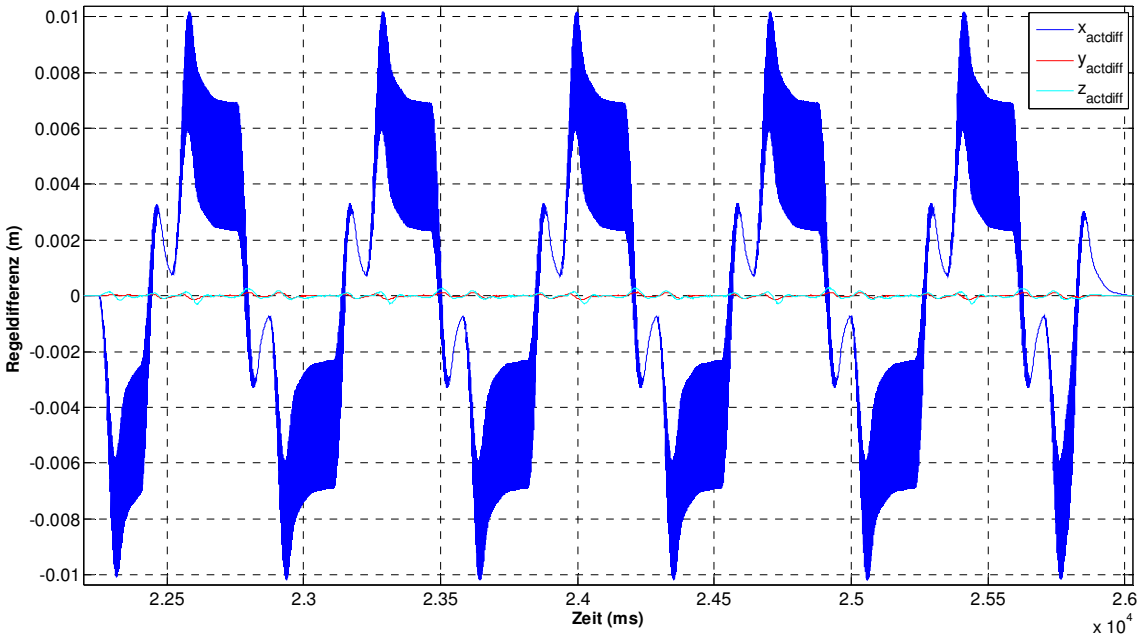

Abbildung 5-9: Regelfehler in der X-Richtung bei 1000 Hz Regelungsfrequenz, Geschwindigkeit 4,6 m/s, Beschleunigung 100m/s^2

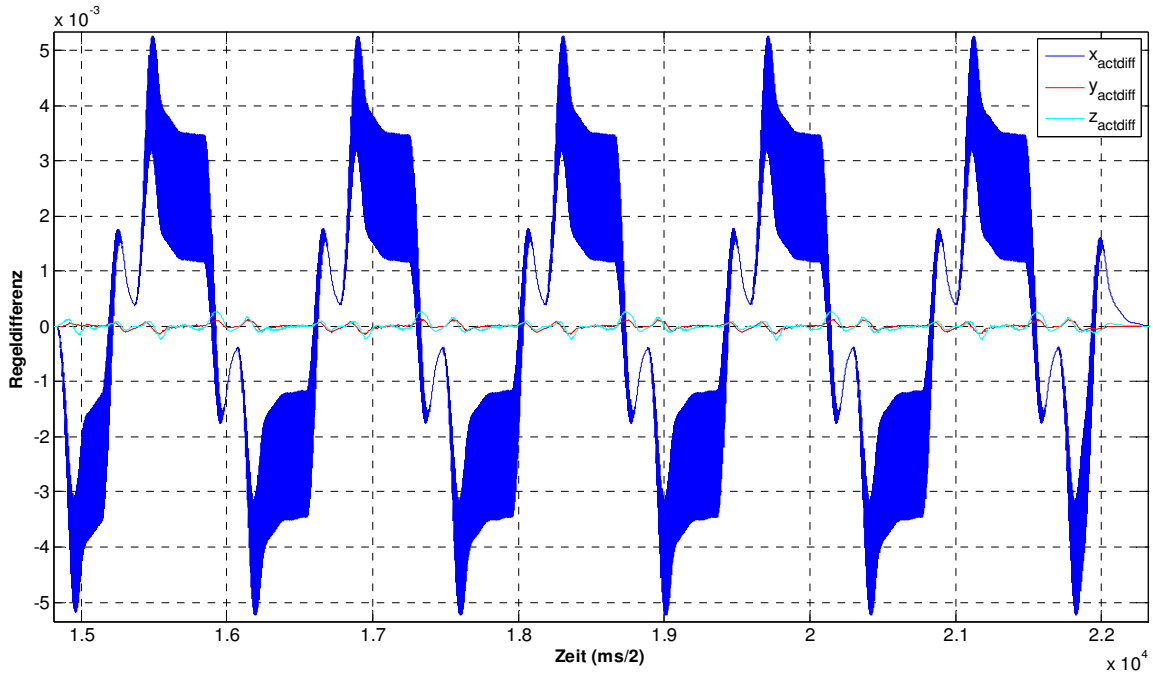

Abbildung 5-10: Regelfehler in der X-Richtung bei 2000 Hz Regelungsfrequenz, Geschwindigkeit 4,6 m/s, Beschleunigung 100m/s^2

5.5 Zusammenfassung

Durch die Optimierung der Kommunikationsmodule konnte eine erhebliche Steigerung der gesamten Systemperformanz erreicht werden. Die zyklische Datenverarbeitungszeit, die innerhalb eines Kommunikationsmoduls zwischen dem Eingang eines MDT und dem Versand des nächsten DDT gemessen wird, konnte um den Faktor 32 von 480 µs auf 15 µs verbessert werden. Dadurch konnte der zeitliche Anteil der Kommunikation im zyklischen Betrieb von ca. 640 µs auf 60 µs reduziert werden. Mit diesem Ergebnis lassen sich theoretisch Steuerungsfrequenzen bis 8 kHz einwandfrei realisieren.

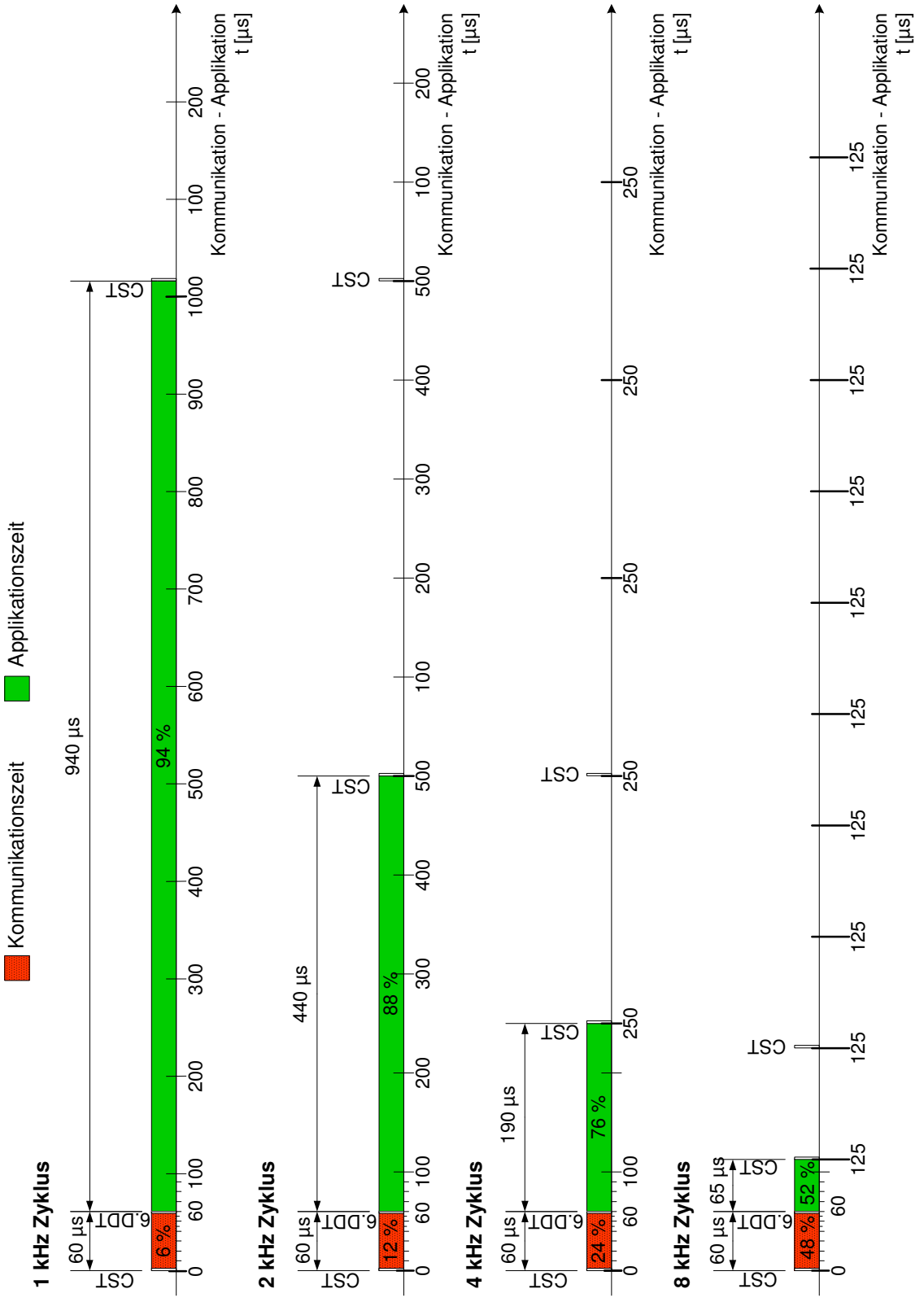

Abbildung 5-11: Zyklusaufteilung in Kommunikations- und Applikationszeit

6 Verteiltes System

Nachdem in den vorigen Kapiteln die Optimierung der Kommunikations-Infrastruktur, inklusive der Funktionalitäten der Middleware MiRPA-X, behandelt wurde, erfolgt in diesem Kapitel, zwecks Erweiterung der für Anwenderapplikationen zur Verfügung stehenden Rechenleistung, die Beschreibung der Entwicklung und Implementierung der verteilten Version von MiRPA-X. Im Sonderforschungsbereich SFB562 soll weiterhin ein zentrales PC-basiertes Steuerungskonzept umgesetzt werden, das allerdings zur Berechnung aufwendiger Algorithmen auf die Ressourcen verteilter Rechner zurückgreifen kann. Das aufgebaute verteilte System besteht aus einem Rechnerverbund mit kleiner räumlicher Ausdehnung mit bis zu 10 über den IEEE1394 Standard verbundenen Rechnern. Die entstehende verteilte Middleware MiRPA-XD[18] ermöglicht der Anwenderapplikation eine transparente Nutzung von Kommunikationsmechanismen zur Ausführung von Funktionseinheiten auf unterschiedlichen Rechnern. Bei der Entwicklung wurde unter anderem das Ziel verfolgt, die Realisierung hoch performanter und echtzeitfähiger Rechenprozessen im verteilten Netzwerk zu unterstützen. Aus diesem Grund wurde von dem in [Koh07] empfohlenen Verteilungsmuster abgewichen, das über das IAP-Protokoll hätte realisiert werden sollen. Zur Vereinfachung wird die verteilte Middleware MiRPA-XD im weiteren Verlauf dieser Arbeit als XD bezeichnet.

6.1 Motivation

Um den wachsenden und dynamisch wechselnden Rechenleistungsbedarf abzudecken, der durch immer komplexer und zeitintensiver werdende Steuerungsstrategien (mit Berücksichtigung von Singularitätsberechnungen [HMB05] [BMC+05], Integration von Sichtsystemen [HHC96]) bedingt wird, muss die verfügbare Rechenleistung dynamisch erweiterbar sein. Um dieses Ziel zu erreichen, kann sowohl ein Mehrkernprozessor als auch ein Rechnernetzwerk eingesetzt werden. Ein Vergleich beider Ansätze wurde in [Kal10] gemacht. In Bezug auf die Inter-Prozessor-Kommunikation unterscheiden sich beide Ansätze erheblich. Mehrkernprozessoren bieten meistens eine um 1-2 Größenordnungen höhere Kommunikationsbandbreite mit um mehrere Größenordnungen kleineren Latenzen. Weiterhin haben Mehrkernprozessoren eine feste Anzahl von Prozessorkernen, deren Rechenleistung nicht dynamisch erweiterbar ist. Auf der anderen Seite weist ein Rechnernetzwerk eine durch den Netzwerk-Kommunikations-Overhead bedingte geringere Bandbreite auf. Es bietet allerdings mehr Ressourcen (z.B. Speicher, IO-Interface, ...) und ermöglicht ein übersichtliches und flexibles System-Design, in dem besonders rechenaufwendige Steuerungsfunktionalitäten (z.B. Aufbereitung von Kamera-Daten) dedizierter Rechner zugeordnet werden können. Darüber hinaus ist die verfügbare Rechenleistung in einem Rechnernetzwerk skalierbar.

[18] MiRPA-X Distributed

XD unterstützt die Entwicklung von Anwendungen sowohl auf einer Mehrkernprozessor-Plattform als auch in einem Rechnernetzwerk. Während die Nutzung der Rechenressourcen auf der Mehrkernprozessor-Plattform meistens durch einen entsprechenden Kernel (z.B. QNX-Kernel) verwaltet wird, wird sie im Rechnernetzwerk von der auf einer höheren Softwareschicht liegenden Middleware XD verwaltet.

Die Verteilung eines Rechenprozesses in einem Rechnernetzwerk lohnt sich im Allgemeinen erst, wenn der zeitliche Verteilungsaufwand geringer ausfällt als die zur Ausführung des Rechenprozesses auf dem lokalen Rechner benötigte Rechenzeit. Daraus folgt, dass eine verteilte Ausführung nur für rechenaufwendige Applikationsmodule sinnvoll ist. Der Rechenaufwand ist in diesem Zusammenhang im Bezug auf den Steuerungszyklus relativ zu sehen. Bei einem Steuerungszyklus mit einer Frequenz von 1 bis 2 kHz werden Applikationsmodule mit 100-200 µs Rechenzeit als rechenaufwendig eingestuft, während sie es bei einer Frequenz kleiner 500 Hz nicht wären. Der Verteilungsaufwand hängt direkt mit dem zeitlichen Kommunikationsaufwand (inklusive Protokollstack, Paketübertragungszeit) des unterlagerten Bussystems zusammen. Daraus resultiert die erste Anforderung an das Bussystem des Rechnernetzwerks, das die Realisierung von Kommunikationsvorgängen mit kleiner Latenz ermöglichen muss. Bei Netzwerkzugriffen mit maximal 100 Bytes Nutzdaten (wie bei Steuerungs-Softwaremodulen für parallele Kinematiken üblich) sollte der Kommunikationsaufwand weniger als 100 µs für Anfrage und Antwort betragen. Ferner soll das eingesetzte Bussystem deterministische Übertragungsmechanismen bereitstellen, die die Realisierung echtzeitfähiger verteilter Anwendungen ermöglichen.

Der Markt und die Literatur bieten zahlreiche Ansätze für die Realisierung von Rechnernetzwerken, die als verteilte Systeme in die PC-basierten Steuerungstechnologien eingesetzt werden. Eine kleine Auswahl ist durch folgende Frameworks gegeben: MiRPA [FKK+07], aRD [BH06], OROCOS[ORO09], ORCA[BKM+05], OSACA [PW97], MIRO [USE+02], PLAYER [VGH03] und RT-CORBA [RJ98], MARIE [CLM+04]. Diese Frameworks setzen alle auf die Ethernet-Technologie als Kommunikationsmedium in der physikalischen Schicht des ISO/OSI-Referenzmodells. Im Bezug auf die zu erreichende Echtzeitfähigkeit im verteilten System treten keine Kollisionen mehr auf dem Ethernet-Bus auf, wenn der Vollduplexmodus betrieben wird. Dennoch wirken die dafür eingesetzten Switches als Verzögerungsglieder auf dem Übertragungsweg. Die daraus resultierende Verzögerungszeit hängt mit dem aktuellen Kommunikationsaufkommen und den Telegrammgrößen zusammen; sie ist daher nicht vorhersagbar. Aufgrund dieser Unvorhersagbarkeit kann trotz Vollduplex keine Deterministik im verteilten System garantiert werden. Ferner setzen die meisten oben genannten Frameworks auf Standard Protokolle wie TCP und UDP. Einige Lösungen (MiRPA, aRD) setzen auf proprietäre Protokolle wie QNET von „QNX Software Systems“. Der durch die Protokoll-Implementierung verursachte Overhead wirkt sich negativ auf die Performanz der verteilten Anwendungen aus, so dass oben genannte Anforderungen kaum erfüllt werden können.

Um ein geeignetes System für die Entwicklung verteilter Steuerungs-Softwaremodule für parallele Kinematiken zur Verfügung zu stellen, wurde im Rahmen des SFB562 ein auf dem IEEE1394 Standard [And99] basierendes verteiltes System entwickelt. Dabei wurde XD als Middleware-Framework eingesetzt, das das verteilte System für den Anwender transparent macht. Das von XD gestützte Framework sollte folgende Eigenschaften aufweisen:

- **Einfache Programmierbarkeit**: das Framework muss einfache Programmierungsmuster und Schnittstellen zur Verfügung stellen.
- **Deterministik**: Kommunikationsmechanismen und Interface-Funktionen müssen eine maximale (Worst Case) Latenz aufweisen.
- **Performanz**: Echtzeitsysteme sind nicht "schnelle" Systeme, sondern "voraussagbare" Systeme von einem TIMING-Gesichtspunkt betrachtet. Dennoch ist die Performanz eine wichtige praktische Größe in vielen Anwendungen. Wenn das beste Programmierungsmuster eine schlechte Performanz aufweist, ist es in der Praxis nicht tauglich.
- **Skalierbarkeit**: das verteilte System muss die Fähigkeit besitzen, dynamisch mit den Rechenanforderungen zu wachsen. Zusätzliche Rechner müssen zur Laufzeit in das System integrierbar sein.

6.2 Diskussion der Netzwerkstruktur

Das Design der verteilten Architektur wurde hauptsächlich durch die gewünschte Echtzeitfähigkeit der verteilten Komponenten der Steuerungssoftware beeinflusst. Eine verteilte Softwarekomponente ist dabei echtzeitfähig, wenn sie, unabhängig vom aktuellen Kommunikationsaufkommen, korrekte Ergebnisse unter Einhaltung gegebener Zeitbedingungen liefert. Bei Steuerungssystemen liegen solche Zeitbedingungen im Bereich einiger Zehn Mikrosekunden. Das ursprüngliche Design [Koh07] der verteilten Middleware XD schlug vor, die Netzwerkverteilung über eine Erweiterung des IAP-Protokolls zu realisieren. Die entsprechende Master/Slave-Architektur ist in Abbildung 6-1a dargestellt. In diesem Ansatz sind sämtliche Netzwerkteilnehmer (Master-Steuerungs-PC, Slave-PC, Sensor-/Aktor-Module) an einem einzigen Bus angeschlossen. Neben dem zyklischen und deterministischen Datenaustausch zwischen dem Steuerungsrechner und den Sensor- und Aktor-Modulen muss das IAP die Kommunikation zwischen Master- und Slave-Rechnern unterstützen. Theoretisch ist dies möglich, denn es steht genügend Bandbreite zur Verfügung auf dem Bus; weniger als 10% der Bandbreite wird aktuell für die zyklische Kommunikation beansprucht.

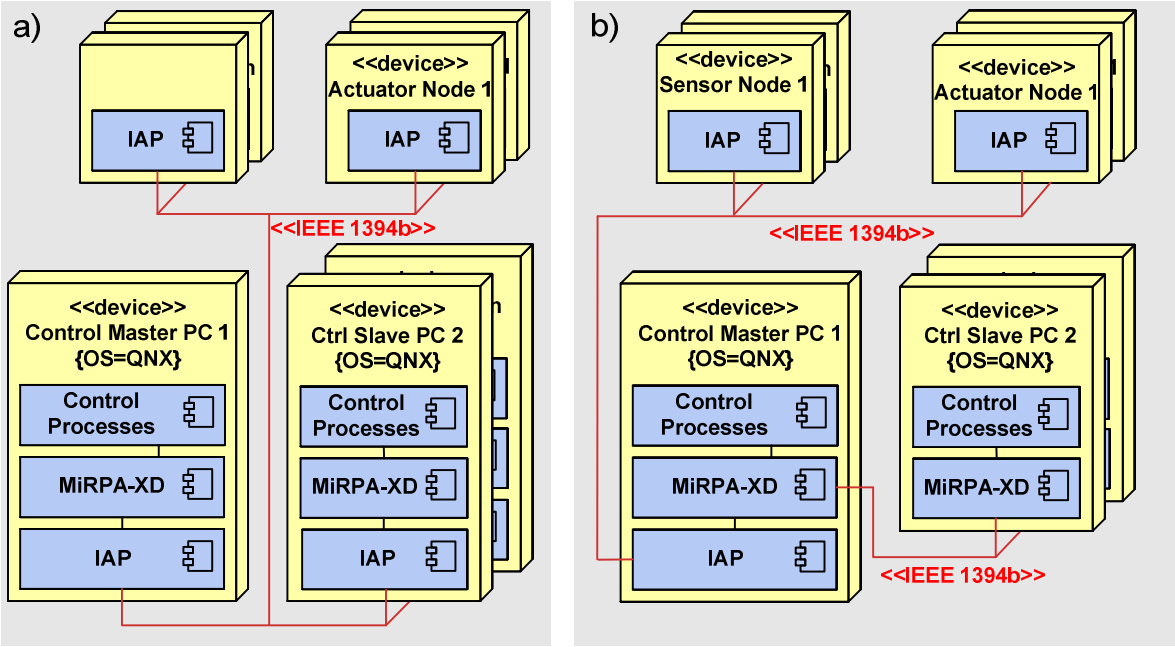

Abbildung 6-1: Netzwerkstruktur der XD Verteilung, a) Verteilung über dem IAP-Protokoll mit einer einzigen Busverbindung, b) Verteilung über einer sekundären Busverbindung

Aber in der Praxis bedeutet dies die Integration einer ereignisbasierten asynchronen Kommunikation in ein zeitgesteuertes Kommunikationsprotokoll. In diesem Fall sind Konflikte in der Protokoll-Schicht vorprogrammiert. Ein solcher Konflikt tritt beispielsweise auf, wenn eine von einem Slave-Rechner initiierte Datenübertragung die Übertragung zeitkritischer Soll- oder Istwerte verzögert. Um dieses Konfliktpotenzial zu beseitigen, kann eine Management-Komponente in das IAP-Protokoll integriert werden, die die Erhaltung der Echtzeiteigenschaften der zyklischen Kommunikation durch Zuweisung priorisierter Kommunikationskanäle gewährleistet. Eine solche Lösung wäre aber nicht optimal, denn sie würde nur die Übertragungsverzögerungen auf die Master/Slave Kommunikation verlagern.

Eine bessere Lösung lässt sich durch die Realisierung physikalisch getrennter Kommunikationswege für die zyklische Kommunikation und die Master/Slave-Kommunikation erreichen. Die daraus entstehende Struktur mit zwei getrennten vom Master-Rechner ausgehenden Bus-Verbindungen ist in Abbildung 6-1b dargestellt. Dort sind Master- und Slave-Rechner logisch über XD-Komponenten miteinander verbunden. Zentrale Komponenten der Steuerungssoftware werden auf dem Master-Rechner ausgeführt, während rechenintensive Algorithmen in Server-Applikationen gekapselt und zur Ausführung auf den Slave-Rechnern ausgelagert werden.

6.3 Auswahl der Bustechnologie

Um echtzeitfähige Mechanismen im verteilten System bereitstellen zu können, müssen die einzelnen Rechner mit einer geeigneten echtzeitfähigen Bustechnologie zusammengeschlossen werden. Die folgende Liste stellt einige Anforderungen an Bustechnologien zusammen, die zu ihrer Bewertung herangezogen werden können:

- Realisierbare ereignisbasierte Kommunikation
- Kommunikationslatenz
- Deterministischer Datentransfer über das Netzwerk
- Zuverlässigkeit (Fehlerkorrektur)
- Datenrate

Neben dem bereits bekannten und eingesetzten IEEE1394 Standard werden einige der für Motion Control-Applikationen eingesetzten Industrial Ethernet Ansätze auf ihre Eignung untersucht und bewertet. Die Kommunikationslatenz gibt an, wie schnell eine Applikation Daten von einem Rechner A zu einem Rechner B transportieren kann. Aufgrund der im Abschnitt 6.1 ausgeführten Argumente ist der Standard Ethernet hierfür ungeeignet.

PROFINET [PM05] teilt die Kommunikationsbandbreite in drei Kanäle auf: den NRT[19]-, den RT[20]- und den IRT[21]-Kanal. Mit dem Einsatz von PROFINET IRT [Pry08] lässt sich theoretisch ein deterministisches Verhalten in einem verteilten Rechnerverbund realisieren. Dafür können Master und Slave-Rechnern dedizierte Slots zugewiesen werden, innerhalb deren sie ihre Daten übertragen. Dennoch ist bei der Realisierung von ereignisbasierter asynchroner Kommunikation mit Reaktionszeiten zu rechnen, die im schlimmsten Fall bis zu einem Kommunikationszyklus (1 ms) andauern. Dies ist der Fall, wenn Daten zu einem Zeitpunkt übertragen werden sollen, wo keine weiteren Slots in dem aktuellen Zyklus zur Verfügung stehen. Einen weiteren Nachteil stellt das geplante Übertragungsverfahren von PROFINET IRT dar. Für die Nutzung des IRT-Kanals muss der gesamte Datenverkehr in der Konfigurationsphase durchgeplant werden. Quell- und Zielknoten sowie die zu übertragende Datenmenge müssen für jeden Verbindungspfad spezifiziert werde. Daher ist es als Basis für die Implementierung ereignisbasierter Kommunikation unflexibel und ungeeignet.

[19] Non Real Time

[20] Real Time

[21] Isochronous Real Time

Ähnlich wie bei PROFINET IRT stellt der Echtzeitmechanismus SCNM[22] von **Ethernet/Powerlink** [Eth03] [SVZ09] ein vorab zu planendes Übertragungsverfahren dar und ist damit ungeeignet.

EtherCAT [Pry08] bietet eine gute Basis für die Realisierung ereignisbasierter Kommunikation. Mit der logisch aufgebauten Ring-Topologie lassen sich Kommunikationszyklen bis 31 µs Zyklusdauer realisieren. Für die Prozessdatenkommunikation geht ein einziges Telegramm zyklisch durch das Netzwerk. Dieses enthält ein vollständiges Abbild der Netzwerkkommunikation mit reservierten Speicherbereichen für Ein- und Ausgabedaten für jeden Slave. Das Telegramm wird vom Master erzeugt und durchläuft nacheinander alle in Reihe geschalteten Slaves, um am Ende wieder in den Master zu gelangen. Während des Durchlaufs durch jeden Slave liest dieser die für ihn bestimmten Daten heraus und schreibt seinerseits ausgehende Daten in entsprechende Speicherbereiche hinein. Dank des Einsatzes spezieller ASICs[23] erfolgt der Datenaustausch in dem Slave quasi *on the fly*. Dabei erfährt das Telegramm eine Verzögerung von ca. 60 ns. Wegen des zyklischen Datentransfers kann bei der Realisierung einer ereignisbasierten Kommunikation im verteilten Rechnernetzwerk ein maximaler Jitter von 31 µs auftreten. Dieses entsteht beispielsweise, wenn Nachrichten erst nach einem Zyklus auf den Bus übertragen werden, nachdem die Applikation den Versand getätigt hat. Ein Nachteil für den Einsatz von EtherCAT ist der Aufwand, der bei Integration und Portierung einzelner Software-Stacks auf die mit entsprechendem Betriebssystem ausgerüstete Ziel-Plattform entsteht.

Das Funktionsprinzip von **SERCOS III** ähnelt dem von EtherCAT. Auch hier werden Telegramme im Gerätedurchlauf durch die Teilnehmer gelesen bzw. geschrieben. Anders als bei EtherCAT handelt es sich hierbei jedoch nicht um einen zusammenhängenden Speicherbereich, der in einzelne Ethernet-Pakete verpackt wurde, sondern um einzelne Datentelegramme. Die Echtzeitkommunikation wird durch den Einsatz des erprobten zeitschlitzbasierten Kommunikationsprotokolls gewährleistet. Kommunikationszyklen bis 31 µs Zykluszeit lassen sich realisieren. Im Gegensatz zu EtherCAT werden NRT-Daten nicht in einem Speicherbereich getunnelt, sondern nach Ablauf der Echtzeitkommunikation in einem Zeitfenster vordefinierter Länge als reguläre Telegramme verschickt. Mit SERCOS III lässt sich theoretisch eine echtzeitfähige ereignisbasierte Kommunikation im verteilten Rechnernetzwerk aufbauen. Die dabei erreichbaren Latenzen sind mit der von EtherCAT vergleichbar. Außerdem muss hier wie bei vorher beschriebenen Bustechnologien Integrationsaufwand betrieben werden.

Der **IEEE1394** Standard [IEE08] besitzt aufgrund seiner Architektur ein inhärentes deterministisches Verhalten. Die Kommunikationsbandbreite ist in einen isochronen und einen asynchronen Kommunikationskanal aufgeteilt. Auf beiden Kanälen lassen sich echtzeitfähige

[22] Slot Communication Network Management

[23] Application Specific Integrated Circuit

ereignisbasierte Kommunikationsmechanismen im verteilten Rechnernetzwerk implementieren. Bei der Nutzung des isochronen Kanals kann jedoch ein Jitter von bis zu 100 µs im Nachrichtenversand auftreten. Dies ist der Fall, wenn die Applikation eine Nachricht versenden möchte, unmittelbar nachdem der isochrone Kanal 20 µs nach Zyklusstart geschlossen wurde und die Nachricht erst im nächsten Buszyklus (nach maximal 100 µs) versendet werden kann. Außerdem macht das ähnlich wie bei PROFINET und Powerlink eingebaute geplante Übertragungsverfahren die Nutzung des isochronen Kanals ungeeignet. Der asynchrone Kanal hingegen eignet sich für die Realisierung ereignisbasierter Kommunikationsmechanismen. Das neue Arbitrierungsverfahren BOSS[24] sorgt nicht nur dafür, dass keine Kollisionen auftreten, sondern auch für einen kürzeren Abstand (durchschnittlich 2 µs) zwischen aufeinanderfolgenden Paketen auf dem Bus und für folglich eine bessere Ausnutzung der Bandbreite, die der ereignisbasierten Kommunikation zugute kommt. Die Vorteile des BOSS-Verfahrens kommen daher, dass dabei die Arbitrierung parallel zum regulären Paketversand durchgeführt wird. Im asynchronen Kanal unterstützt der IEEE1394 eine prioritätsbasierte Datenübertragung. Die Teilnehmer-Prioritäten werden aus der Netzwerktopologie generiert. Dank des Fairness-Verfahrens wird vermieden, dass ein Teilnehmer den Bus dauerhaft besetzt. In einem Fairness-intervall wird im normalen Betrieb jedem Teilnehmer der Buszugriff ein einziges Mal gewährt. Dennoch können Teilnehmer mit hoher Buslast durch eine Priorisierungsfunktionalität so konfiguriert werden, dass ihnen ein mehrfacher Buszugriff innerhalb eines Fairness-Intervalls eingeräumt wird.

Tabelle 2: Bewertungsergebnis von echtzeitfähigen Bussystemen

	Profinet IRT	EtherCat	SERCOS 3	PowerLink	IEEE1394
ereignisbasierte Kommunikation	-	+	+	-	+
Kommunikationslatenz	+	+	+	+	+
Deterministischer Datentransfer	+	+	+	+	+
Datenrate	+	+	+	+	+
Wirtschaftsfaktor	-	-	-	-	+

Das Ergebnis der oberen Bewertung wird in Tabelle 2 dargestellt. Von den untersuchten Bussystemen können sowohl EtherCAT, als auch SERCOS III und der IEEE1394 Standard für den Aufbau von ereignisbasierter Kommunikation im Rechnernetzwerk eingesetzt werden. Was aber die Performanz angeht, ist der IEEE1394 wegen seinem echtzeitfähigen asynchronen Kanal und dem minimalen Jitter das bessere System. Außerdem bieten die bereits in den vorherigen Arbeiten [Koh07] mit der Entwicklung des IEEE1394-Software-Stack gewonnenen Erfahrungen einen optimalen Ausgangpunkt für den Einsatz des IEEE1394 Standards in das verteilte System.

[24] Bus Owner Supervisor Selection

Für den Einsatz des IEEE1394 Standards spricht auch der wirtschaftliche Faktor, denn die entsprechenden PCI-Hardware-Komponenten sind kostengünstig erhältlich.

6.4 Aufbau der verteilten Middleware

XD wird als Middleware ausgelegt, die den Verteilungscharakter des Systems vor dem Anwender verbirgt. Es weitet die Anwendung des Message-Passing basierten Nachrichtendienstes von MiRPA-X auf das verteilte System aus. In der Client-Anfrage wird weder der Ziel-Server angegeben, der die Anfrage bearbeitet, noch der Zielrechner spezifiziert, auf dem der Server ausgeführt wird. Im verteilten System unterscheidet XD zwischen Master- und Slave-Rechnern (Abbildung 6-2a). Client- und Server-Applikationen dürfen sowohl auf dem XD-Master als auch auf den XD-Slaves implementiert werden. Um den Datenfluss im verteilten System besser steuern zu können, wurde jedoch die Einschränkung eingebaut, dass ausschließlich auf dem XD-Master laufende Client-Applikationen eine Netzwerkanfrage initiieren dürfen.

In Abbildung 6-2b werden die Softwarekomponenten von XD dargestellt. Neben dem im Abschnitt 2.3 bereits erwähnten ObjectServer (OBS) und Token Scheduler (TS) wurde die Middleware um die Komponenten Remote Control (RC) und System Monitor (SM) erweitert. All diese Komponenten wurden in eigenständigen miteinander synchronisierten Threads implementiert. Der RC ist für die Erweiterung der Message-Passing-Kommunikation über die physikalische Rechnergrenze hinweg zuständig. Außerdem bietet er die Mechanismen, die die automatische Integration neuer Rechner in das verteilte XD-Netzwerk zur Laufzeit ermöglichen. Der SM hat die Aufgabe, ein Abbild der aktuellen Systemaktivitäten zur Laufzeit für den Anwender zu generieren. Er sammelt Topologie-Informationen und Task-Ausführungszeit und hält sie in einer Datenbank bereit, die als Plattform für die Integration von Self-X-Eigenschaften in die Robotersteuerung dient. Eine detaillierte Ausführung des System Monitors findet im Kapitel 7 statt.

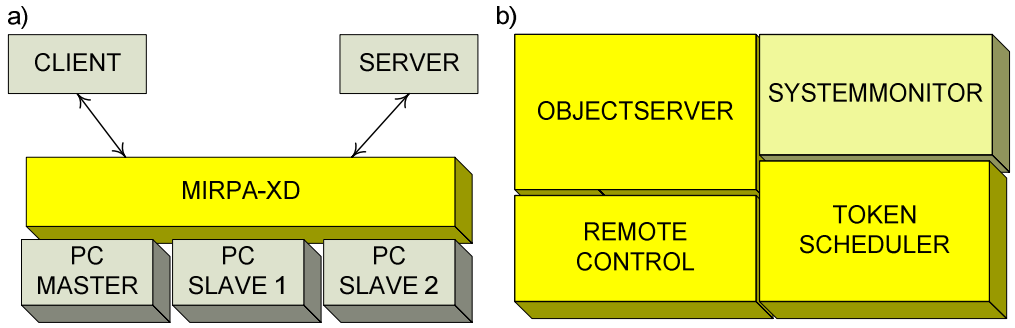

Abbildung 6-2: Übersicht des MiRPA-XD Designs

6.5 Dynamische Integration zusätzlicher Slaves im laufenden Betrieb

Um die Skalierbarkeit des verteilten Systems zu gewährleisten, unterstützt XD die Integration neuer Slave-Rechner im laufenden Betrieb [DMM09]. Die durch das Plug-in eines Slaves ausgeführten Prozessabläufe sind in Abbildung 6-3 dargestellt. Nachdem ein neuer Slave-Rechner in das Netzwerk eingefügt wird, löst der FireWire Bus einen Bus-Reset aus, der die Re-Initialisierung der Bus-Topologie einleitet. Da ein separates Bussegment für die Master/Slave-Kommunikation eingesetzt wird (Abbildung 6-1), bleibt die zyklische Echtzeitkommunikation zwischen dem Steuerungs-PC und den Sensor/Aktoren-Einheiten ungestört. Nach dem Bus-Reset wartet der Master-Rechner in einem blockierten Zustand, bis eine vordefinierte Initialisierungszeit (IT) abläuft. Während dieser Zeit schließen sämtliche am Netzwerk angeschlossenen Slaves den IEEE1394 Re-Initialisierungsprozess ab und erhalten allesamt eine neue Teilnehmer-Identifikationsnummer (ID). Anschließend wechseln sie in den aktiven Zustand, in dem sie Topologie-Informationen mit dem Master austauschen können. Die Topologie-Informationen bestehen aus dem FireWire spezifischen 48 Bit breiten *Global Unique Identifier* (GUID), der jeden FireWire-Controller-Chip eindeutig identifiziert, und dem in der Initialisierungsphase erhaltenen ID.

Nach Ablauf der IT-Zeit sendet der Master eine Konfigurationsnachricht an alle Slaves mit der Aufforderung, deren Topologie-Informationen zu übermitteln. Mit der Konfigurationsnachricht übermittelt der Master gleichzeitig eigene Topologie-Informationen, so dass jeder Slave daraufhin eine Punkt-zu-Punkt Verbindung mit dem Master aufbauen kann. Nach dem Versand der Konfigurationsnachricht wartet der Master erneut blockiert ab, bis die zur Topologie-Generierung vorgesehene Zeit (TGT) abläuft. Die TGT-Zeit ist so ausgelegt, dass innerhalb dieser sämtliche Slaves (inklusiv dem neuen eingefügten Slave) ihre Topologie-Informationen an den Master senden können. In der Praxis dauert der Vorgang ein paar hundert Mikrosekunden. Nach Erhalt der Slave-Daten erstellt der Master ein lokales Abbild der gesamten Netzwerk-Topologie und wechselt in den aktiven Zustand am Ende der TGT-Zeit.

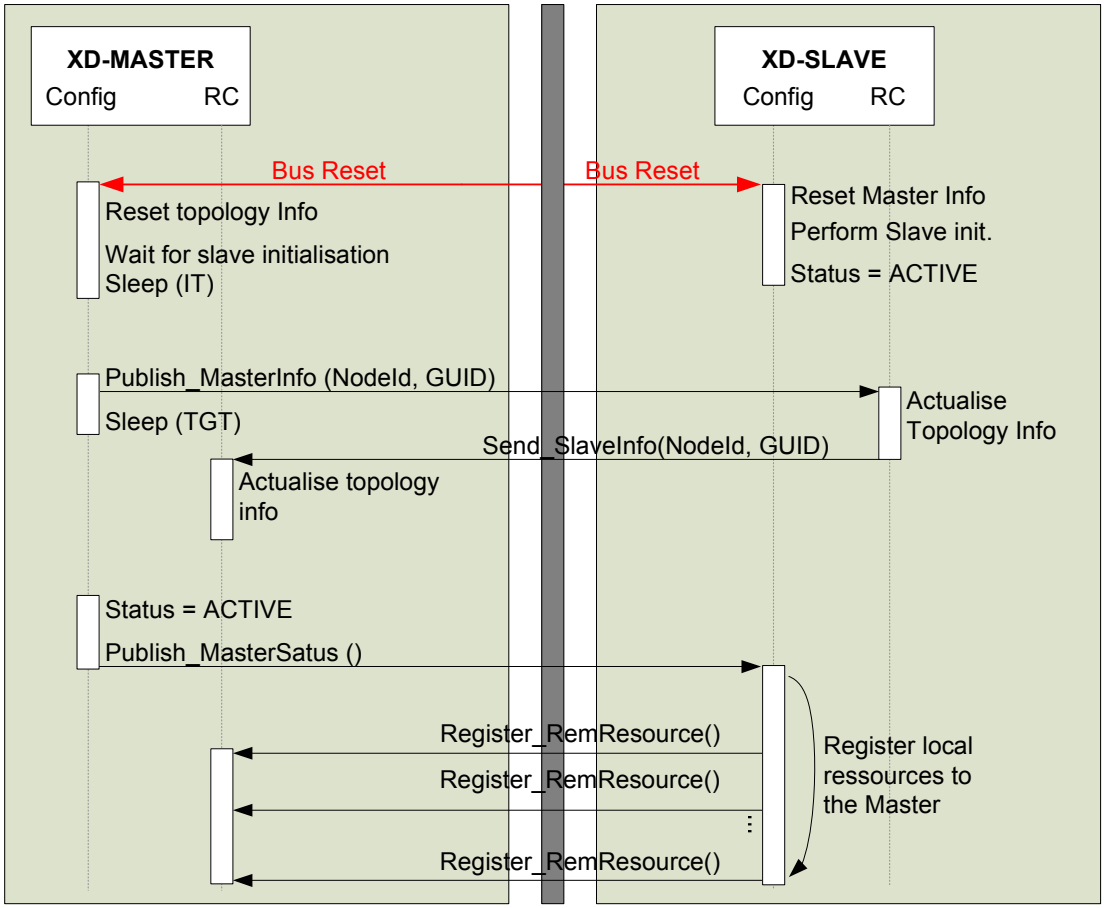

Abbildung 6-3: Automatische Netzwerk-Konfigurationssequenz nach Plug-in eines Slave-Rechners

Ab diesem Zeitpunkt kann die automatische Registrierung von entfernten Slave-Ressourcen beginnen. Nach Erhalt einer entsprechenden Statusmeldung vom Master meldet der neue Slave sämtliche lokalen nachrichtenbasierten Ressourcen bei dem Master an. Der Registrierungsvorgang ist in Abschnitt 6.6 näher erläutert. Nach Abschluss der Registrierung weiß der Master, welche Ressourcen auf welchem Slave ausgeführt sind, und kann während der Laufzeit gegebenenfalls auf sie zugreifen. Die mit der dynamischen Integration zusätzlicher Slaves in das Netzwerk verbundenen Prozessabläufe haben keinen Einfluss auf die Echtzeitfähigkeit der Steuerungsapplikationen auf dem Master-Rechner, da sie nebenläufig in einem Thread bei niedriger Priorität ausgeführt werden.

6.6 Registrierung entfernter Ressourcen

XD verwaltet den Zugriff auf Ressourcen in der verteilten Umgebung mittels eines Master/Slave-Modells, in dem Hauptsteuerungskomponenten auf dem Master-Rechner ausgeführt werden und rechenaufwendige Prozesse zur Ausführung auf Slave-Rechner verteilt werden. Ein Vorteil des Master/Slave-Modells ist, dass der Master sämtliche Zugriffe koordiniert

und als zentrale Komponente fungiert, um die das ganze System aufgebaut wird. Dadurch wird die Systementwicklung vereinfacht. Bei dem Master/Slave-Modell muss die auf dem Master laufende XD-Instanz (XD-Master) wissen, welche Slaves als Teilnehmer am Netzwerk angeschlossen sind und vor allem, welche Ressourcen von welchem Slave zur Verfügung gestellt wird oder welcher Task auf welchem Slave ausgeführt werden kann. Um diesen Zweck zu erfüllen, unterstützt XD die automatische Registrierung von verteilen Ressourcen auf dem XD-Master. Dafür wurde die Registrierungsroutine von XD so erweitert, dass jede Ressourcenregistrierung auf einem XD-Slave automatisch an den XD-Master übermittelt wird.

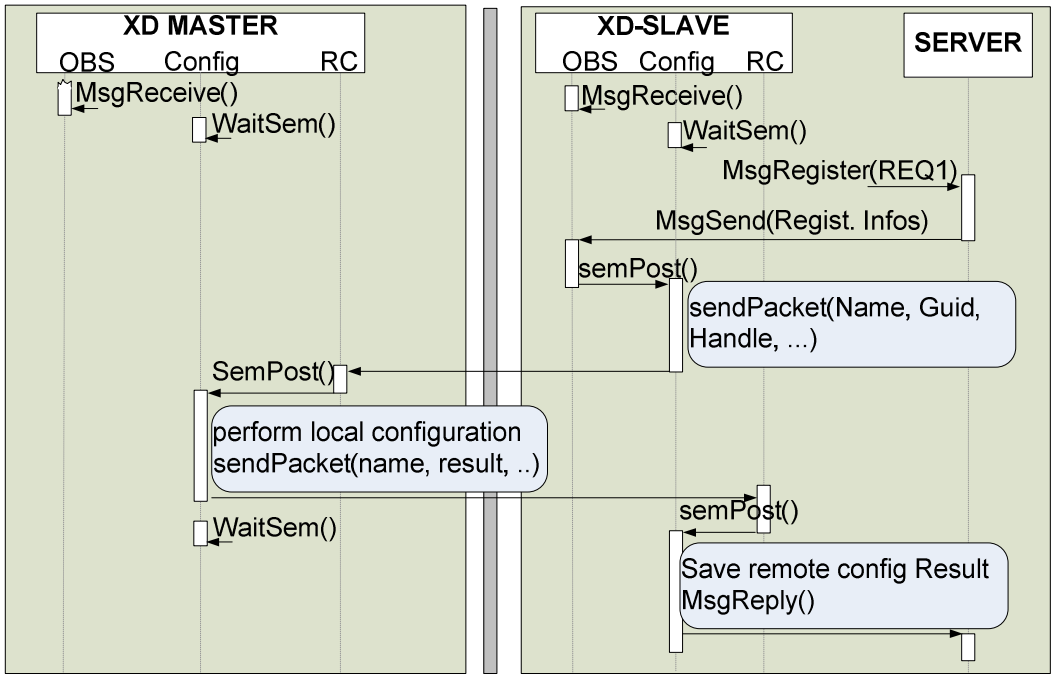

Abbildung 6-4: Registrierung einer entfernten Ressource auf dem Master-Rechner

Abbildung 6-4 beschreibt den erweiterten Registrierungsvorgang einer Ressource auf einem XD-Slave. An dieser Stelle sei angemerkt, dass Ressourcen innerhalb Server-Applikationen als Applikationscode implementiert werden, die als Reaktion auf dem Empfang von COMMAND- und REQUEST-Nachrichten ausgeführt werden. Um die Nachricht REQ1 zu registrieren, sendet der Server eine entsprechende Konfigurationsnachricht an den XD-Slave. Nachdem die Nachricht lokal registriert wurde, sendet der XD-Slave ein IEEE1394-Konfigurationstelegramm an dem XD-Master, um dort die neue lokal registrierte Nachricht bekannt zu machen.

Der Aufbau des Konfigurationstelegramms ist in Abbildung 6-5 dargestellt. Man unterscheidet zwei Typen von Konfigurationstelegrammen: Anfrage-Telegramme (REMOTE-CONFIG-REQUEST) und Antwort-Telegramme (REMOTE-CONFIG-RESULT). Der Typ eines Konfigurationstelegramms wird in dem Bitfeld FUNC festgelegt. Mit dem Bitfeld SUBFUNC wird der Typ des Registrierungsvorgangs festgelegt. Man unterscheidet zwischen dem An– und Abmelden von sowohl COMMAND- und REQUEST-Nachrichten als auch Server-Prozessen.

Außerdem wird im Fall einer COMMAND- oder REQUEST-Nachricht die Prozess-Identifikationsnummer (Sender PID) des Server-Prozesses übermittelt, der die Nachricht registrieren möchte. Mit den zwei GUID-Feldern wird der GUID des Slave-Rechners an den Master gesendet. Anhand des GUID kann der Master die angemeldete Nachricht eindeutig einem Slave-Rechner im verteilten System zuordnen. Mit dem Nachrichten-Handle wird die interne Referenznummer der Nachricht innerhalb der XD-Nachrichtenverwaltungstabelle an den Master gesendet. Außerdem werden maximal jeweils 64 Bytes für den Nachrichtennamen und den zugehörigen Servernamen reserviert.

Nach Erhalt der Konfigurationsnachricht trägt der Master die zu registrierende Nachricht in die interne Nachrichtenverwaltungstabelle ein, mit dem Vermerk, dass die Nachricht nicht zu einem lokalen Server, sondern zu einem entfernten Server im verteilten System gehört. Dabei speichert er Zusatzinformationen über den Ziel-Server, wie GUID des Slave-Rechners und Nachrichten-Handle, damit spätere Nachrichtenübermittlungen zeitoptimal und erfolgreich durchgeführt werden können. Nach Abschluss der Konfiguration sendet der Master ein Konfigurationstelegramm mit dem Konfigurationsergebnis (Abbildung 6-4). Dieses Ergebnis (SUCCESS oder FAILED) wird im Bitfeld RESULT gespeichert. Daraufhin kann der Konfigurationsvorgang auf dem Slave-Rechner abgeschlossen werden. Das Abmelden einer Ressource erfolgt nach einem ähnlichen Schema.

Damit der Slave bei fehlerhaftem Verhalten des Masters während des Konfigurationsprozesses nicht unendlich blockiert, ist eine Timeout-Funktion in die Konfigurationstelegramm-Sende-Funktion integriert. Falls der Master eine Konfigurationsanfrage nach Ablauf des Timers nicht beantwortet hat, merkt dies der Slave und bricht den Konfigurationsprozess mit entsprechender Fehlermeldung an der Serverapplikation ab.

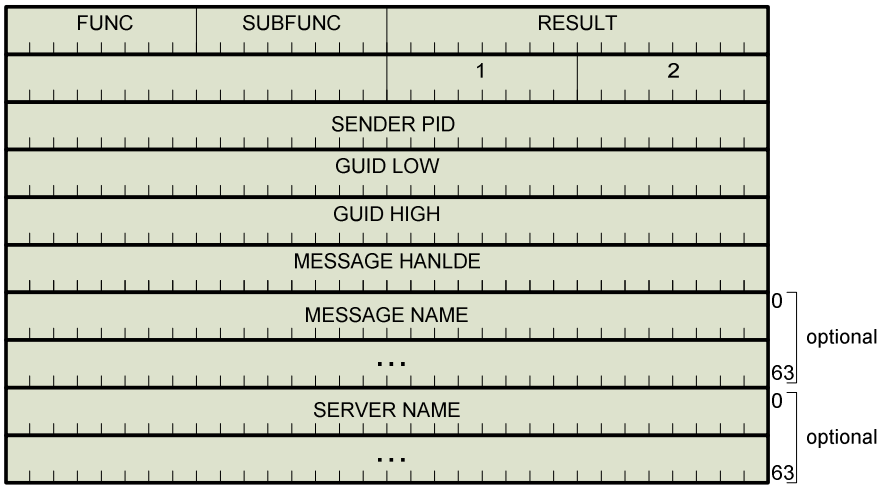

Abbildung 6-5: Format eines XD-Konfigurationstelegramms

6.7 Beenden der XD-Instanzen

Wenn ein XD-Slave versehentlich beendet oder aus dem verteilten Netzwerk entfernt wird, stehen normalerweise sämtliche von ihm registrierten Ressourcen dem XD-Master wegen der physikalischen Trennung nicht mehr zur Verfügung. Das auf dem XD-Master gespeicherte Abbild der registrierten verteilten Ressource stimmt somit nicht mehr mit der realen Verteilung überein. Dieser Fehlerzustand von XD führt zwangsläufig zu Funktionsfehlern bei der Ausführung von verteilten Applikationen.

Um diesen Fehlerzustand zu vermeiden und/oder dessen Auswirkungen auf die verteilten Applikationen zu reduzieren, wurde ein Sperrmechanismus in den XD-Slave integriert. Demnach darf eine XD-Slave-Instanz nur dann beendet werden, wenn sämtliche zuvor registrierte Ressourcen abgemeldet wurden. Dies bedeutet, dass alle Server-Applikationen sämtliche registrierten Nachrichten vor dem Beenden der XD-Slave-Instanz abgemeldet haben müssen. Bei einem unbeabsichtigten Entfernen eines XD-Slaves aus dem verteilten Netzwerk tritt das von FireWire spezifizierte Bus-Reset auf. Infolge des Bus-Resets wird der im Abschnitt 6.5 und Abbildung 6-3 beschriebene Re-Initialisierungsvorgang durchgeführt und das gespeicherte Ressourcen-Abbild des XD-Masters entsprechend aktueller Ressourcenverteilung angepasst.

6.8 Zugriff auf entfernte Ressourcen

Unter XD ist der Zugriff auf entfernte Ressourcen für den Anwender transparent. Client-Anfragen beinhalten weder Details über den Ziel-Server noch Informationen über den Rechner,

auf dem der Server ausgeführt wird. In Abbildung 6-6 wird der Zugriff auf entfernte Ressourcen mit einem vereinfachten Sequenzdiagramm dargestellt. Der Zugriff erfolgt über das auf Netzwerkebene ausgeweitete Message-Passing. Nach Erhalt einer Client-Anfrage entnimmt der XD-Master die Information über den Ziel-Server und dessen Standort aus der internen Nachrichtenverwaltungstabelle. Sollte der Ziel-Server auf dem lokalen Rechner vorhanden sein, dann würde XD die Nachricht wie bisher sofort zustellen. Falls der Ziel-Server sich auf einem Slave-Rechner im Netzwerk befindet, dann wandelt XD das strukturierte Nachrichtendatenobjekt in das für IEEE1394-Netzwerktransaktionen vorbereitete Kommunikationstelegramm um (*Marshalling*).

Der Aufbau des Kommunikationstelegramms ist in Abbildung 6-7 dargestellt. Das Bitfeld FUNC enthält in diesem Fall den festgelegten Typ REMOTE_APP_DATA. In dem Bitfeld SUBFUNC wird der Typ der XD-Nachricht eingetragen. Dabei wird zwischen den Nachrichtentypen COMMAND, REQUEST und REPLY unterschieden. Der Sender PID und der GUID dienen der Identifikation des anfragenden Clients auf dem Ziel-Rechner. Das Feld MESSAGE SPECIFICATION ist nur im Fall einer Nachricht vom Typ REQUEST mit der entsprechenden Information belegt [Koh07]. Um geringe Kommunikationslatenzen zu verursachen, wurden während des Marschalling-Verfahrens nur die nötigen Protokollinformationen in das Kommunikationstelegramm kopiert. Bei 20 Bytes Protokolldaten können bis zu 4 kBytes Applikationsdaten in einem Telegramm zusammengestellt werden. Deren Größe wird im Feld APPLICATION DATA SIZE angegeben.

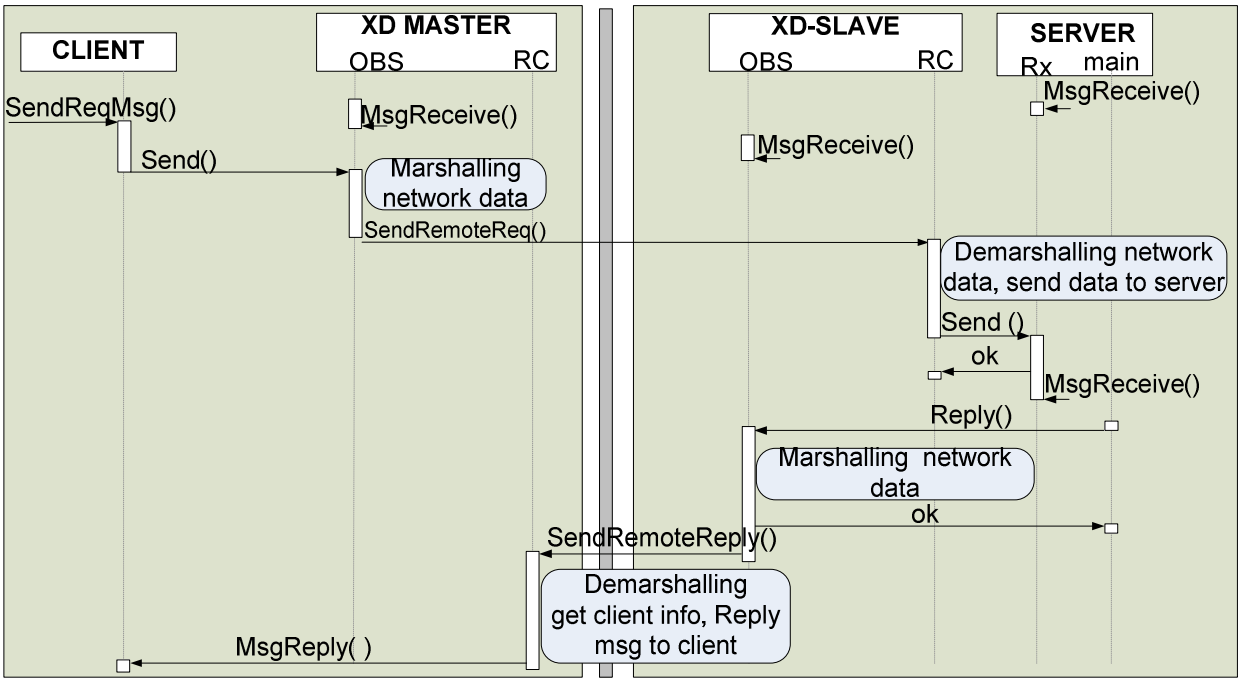

Abbildung 6-6: Zugriff auf entfernte Ressourcen über erweitertes Message-Passing

Nach dem Daten-Marshalling sendet der XD-Master das entsprechende Kommunikationstelegramm an den Slave-Rechner. Nach Erhalt des Telegramms startet der XD-Slave den Demarshalling-Prozess, wandelt die Telegrammdaten in das ursprüngliche Nachrichtendatenobjekt zurück und speichert die Client-Informationen in einer internen Liste ab. Danach stellt er dem Ziel-Server die Nachricht zu. Bei REQUEST-Nachrichten sendet der Server, nach Abarbeitung der Anfrage, eine Antwort an den anfragenden Client. Die Antwort in Form einer REPLY-Nachricht wird erst an den XD-Slave geschickt. Anhand zuvor gespeicherter Client-Informationen kann der XD-Slave die Nachricht an den Master-Rechner weiterleiten. Dort wird anschließend die Antwort-Nachricht an den entsprechenden Client verschickt. Im Gegensatz zu REQUEST-Nachrichten wird bei COMMAND-Nachrichten keine Antwort an den Client verschickt.

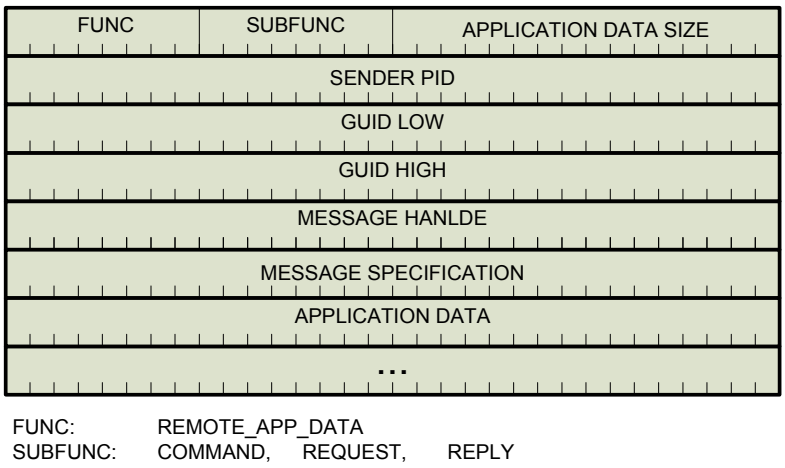

Abbildung 6-7: Format eines XD-Kommunikationstelegramms

6.9 Eingliederung von XD in das ISO/OSI Modell

OSI[25] [ISO96] bezeichnet das Referenzmodell, das von der Internationalen Standardisierung Organisation (ISO) als Grundlage für die Bildung von Kommunikationsstandards erstellt wurde. Das ISO/OSI-Modell unterteilt die Kommunikationsabläufe in sieben Funktionsblöcke oder Schichten, die sich von der physikalischen bis zur Applikationsschicht erstrecken. Aufgrund seiner Netzwerkauslegung sind nicht alle Schichten des OSI-Modells in dem XD-gestützten Kommunikationssystem vertreten. Die Netwerkstruktur sieht keine räumlich weit ausgedehnte Verteilung mit unbegrenzter Teilnehmerzahl vor. Stattdessen ist ein kleines übersichtliches Rechnernetzwerk mit festgelegter maximaler Teilnehmerzahl vorgesehen, in dem der IEEE1394 Standard als Bustechnologie eingesetzt wird. Netzwerkteilnehmer werden anhand während der Businitialisierung gesammelter Topologie-Informationen über eine direkte Verbindung

[25] Open System Interconnection

angesprochen. Ein Netzwerk-Routing ist also nicht notwendig. Aus diesem Grund ist die Netzwerkschicht des OSI-Modells im XD-Kommunikationssystem nicht vorhanden.

Ferner entfällt die Transportschicht im XD-Netzwerk. Ein Grund dafür ist, dass Datensegmentierung aufgrund der in Steuerungsapplikationen üblichen, zwischen Softwaremodulen auszutauschenden kleinen Datengrößen nicht nötig ist. Außerdem sind die Prozeduren zur Fehlerkorrektur in der unteren Sicherungsschicht angesiedelt. Weiterhin fallen die Sitzungs- und Darstellungsschicht aus. Die einzigen vorhandenen Schichten sind die physikalische Schicht, Sicherungsschicht und Applikationsschicht. Dabei sind die Funktionalitäten der physikalischen und der Sicherungsschicht in den IEEE1394-Hardwarekomponenten PHY und LLC integriert.

6.10 Performanz von XD im verteilten System

Um die Performanz von XD in der verteilten Umgebung zu ermitteln, werden zwei identische Rechner eingesetzt. Die Rechner-Hardware besteht aus einem 3 GHz Intel Core 2 Duo E8400 Prozessor, einem 6 MB L2 Cache, 2 GB DDR2 800 Arbeitspeicher, einem Asus P5K64 WS Chipsatz und FireWire 800 Netzwerkadapter. Als Betriebssystem wird die Version 6.3.2 von QNX Neutrino eingesetzt. Als Performanzkriterium wird die Latenz der bidirektionalen Kommunikation, wie sie bei synchronen REQUEST-Mechanismen unter XD erfolgt, festgelegt. Der Messvorgang läuft wie folgt ab: zuerst werden die zwei Rechner als XD-Master und XD-Slave konfiguriert. Während auf dem XD-Slave eine Server-Applikation ausgeführt wird, läuft eine Client-Applikation auf dem XD-Master. Der Client sendet über XD eine Nachricht an den Server, der sofort eine Antwort zurückt sendet. Die Zeit wird im Client unmittelbar vor dem Absenden der Nachricht und nach Empfang der Antwort festgehalten. Aus der Zeitdifferenz wird dann die Latenzzeit ermittelt. Der Messvorgang wird in einer Messreihe 1000-mal wiederholt und ein Mittelwert wird gebildet. Die Messreihen werden für kleine und mittlere Nutzdatengrößen von 10, 50, 100 und 500 Bytes durchgeführt.

Die ermittelte Latenzzeit für die jeweiligen Nutzdatengrößen wird in Abbildung 6-8 dargestellt. Bis 100 Bytes Nutzdaten erreicht XD Latenzzeiten unter 90 µs für die bidirektionale Kommunikation. Ab 200 Bytes übertrifft die Latenzzeit die 100 µs Marke. Bis 100 Bytes ist der maximale Jitter kleiner als 7 µs. Bei 500 Bytes beträgt der maximale Jitter 12 µs. In Abbildung 6-8 werden zum Vergleich die Latenzzeiten dargestellt, die bei direkter Gbit Ethernet-Verbindung (ohne Switch, ohne Middleware-Unterstützung und unter Verwendung des QNET-Protokoll) von Client und Server erzielt werden. Im Vergleich dazu braucht XD nur 10 µs mehr bei der Übertragung von Nutzdaten bis 100 Bytes. Bei 500 Byte beträgt die Differenz maximal 70 µs. Um die Relevanz dieser Ergebnisse zu unterstreichen, muss ein Vergleich mit der von ähnlichen Architekturen erzielbaren Performanz durchgeführt werden. Hierfür werden aus der Literatur [Orc09] [FKW10] ORCA und MiRPA als Vergleichsarchitekturen herangezogen. Laut [Orc09] wurde bei ORCA mit einem 2,13 GHz Intel Core 2 Duo 6400 ein etwas langsamerer Rechner mit einem Gbit-Ethernet-Netzwerkadapter für die Messung eingesetzt. Das ebenfalls in Abbildung 6-8 dargestellte Messergebnis zeigt eine durchgängig schlechtere Performanz

(mindestens 60 µs Zeitdifferenz) im Vergleich zu XD. Im wichtigsten Übertragungsbereich von 100 bis 200 Bytes Nutzdaten ist XD etwa doppelt so schnell wie MiRPA. Erst ab 300 bis 400 Bytes sind beide Systeme in etwa gleich schnell.

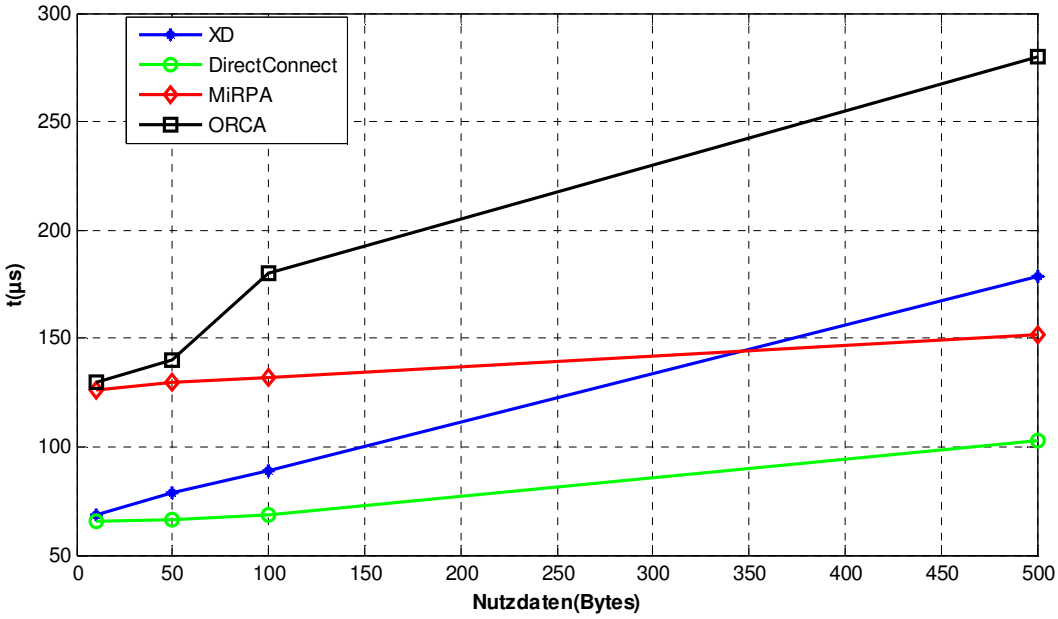

Abbildung 6-8: Latenzzeit der bidirektionalen Kommunikation – Performanzvergleich von XD mit MiRPA und ORCA [Orc09]

6.11 Dienstgüte (QoS)

Die Dienstgüte von XD im verteilten System wird mittels der Latenz der Kommunikationsmechanismen, der Fehlerbehandlung, des deterministischen Verhaltens und der Skalierbarkeit angegeben.

Latenzzeit: wegen der geringen Größe der zwischen Steuerungs-Softwaremodulen auszutauschenden Informationen ist die Übertragungslatenz von Telegrammen kleiner und mittlerer Größen für Steuerungsapplikationen im verteilten System relevant. Wie im letzten Abschnitt bereits gesehen, liefert XD im Vergleich zu gängigen Systemen sehr gute Ergebnisse in diesem Bereich. Die Latenzzeit für die bidirektionale Kommunikation mit bis zu 100 Bytes Nutzdaten beträgt weniger als 90 µs.

Fehlerbehandlung: unter XD werden die bei lokaler Anwendung bereits vorhandenen Fehlererkennungsmechanismen [Koh07] auf das verteilte Netzwerk ausgeweitet. So wird durch die in jeder REQUEST-Anfrage integrierte Watchdog-Timer-Funktion so erweitert, dass anfragende Clients nicht länger als eine spezifizierte maximale Wartezeit blockieren, falls der entsprechende auf einem entfernten Rechner ausgeführte Server nicht antwortet. Ebenso werden

fehlerhafte Client-Anfragen an entfernte Server durch XD mit entsprechender Fehlermeldung zurückgewiesen. Außerdem können EMV-bedingte Übertragungsfehler im verteilten Netzwerk auftreten. XD verlagert die Behandlung solcher Fehler in die Sicherungsschicht der zur Rechnerkopplung eingesetzten IEEE1394-Bustechnologie. In der Tat muss, nach dem IEEE1394 Standard, die Übertragung jedes asynchronen Telegramms durch den Empfänger quittiert werden. Übertragungsfehler werden mit fehlender oder fehlerhafter Quittierung erkannt. Der IEEE1394-LLC behandelt die Übertragungsfehler mittels einer durch Software konfigurierbaren automatischen Sendewiederholung. Dabei stellt die Software ein, wie oft der Sendevorgang maximal wiederholt werden kann, bevor die Sendung als fehlgeschlagen gilt und eine Fehlermeldung an die aufrufende Applikation gesendet wird. Die Sendewiederholung selbst erfolgt ohne Eingriff der Software.

Echtzeitfähigkeit: für Echtzeitsysteme hat das deterministische Verhalten der Kommunikationsvorgänge eine primäre Bedeutung. Hierbei ist gemeint, dass die Kommunikationsvorgänge innerhalb festgelegter Zeiten abgeschlossen werden und keine Pakete im verteilten Netzwerk verloren gehen. XD ermöglicht einen echtzeitfähigen Betrieb im verteilten System mit Hilfe folgender Mechanismen:

- Strukturbedingte Flusskontrolle des Nachrichtenverkehrs: durch die im verteilten System angewandte Master/Slave Struktur werden Anfragen allein vom dem Master initiiert. Dies hat den Vorteil, dass der Master die Busaktivitäten maßgeblich beeinflusst und mit einem gewissen Grad steuert. Durch den Master/Slave-Betrieb kann eine erhöhte Buslast verhindern werden.
- IEEE1394-Mechanismen: dank des BOSS-Arbitrierungsverfahrens treten keine Kollisionen auf dem Bus bei Paketversand auf. Die bei der Arbitrierung geltenden Teilnehmer-Prioritäten werden aus der Netzwerktopologie generiert. Dabei erhält der Master, der gleichzeitig als IEEE1394-Root fungiert, die höchste Priorität. Um zu verhindern, dass ein Teilnehmer (z.B. der Master) den Bus dauerhaft nutzt, setzt der IEEE1394 Standard ein Fairness-Verfahren ein. Innerhalb eines Fairness-Intervalls darf im normalen Betrieb jeder Teilnehmer ein einziges Telegramm über den Bus versenden. Dennoch bietet der IEEE1394 Standard die Möglichkeit, einen Teilnehmer mit absehbarer hoher Buslast so zu konfigurieren, dass er eine definierte maximale Anzahl von Telegrammen innerhalb des Fairness-Intervalls versenden darf.

Skalierbarkeit: unter XD besteht die Möglichkeit, das verteilte System dynamisch mit dem Rechenbedarf wachsen zu lassen. Neue Rechner lassen sich zur Laufzeit in das System integrieren. Dafür müssen sie, auch bei laufenden Anwendungs-Prozessen, einfach an das Netzwerk angeschlossen werden. Daraufhin führt XD automatische eine Netzwerkrekonfiguration durch, und nach einigen hunderten von Mikrosekunden stehen die Ressourcen des neuen Rechners dem zentralen Steuerungsrechner zur Verfügung. Die entsprechenden Integrationsmechanismen wurden in Abschnitt 6.5 erläutet.

6.12 Realisierung der statischen Verteilung von Steuerungskomponenten

Aufgrund des nachrichtenbasierten und modularen Aufbaus der RCA562 Robotersteuerung [Maa09] lassen sich einzelne Steuerungskomponenten mit geringem Aufwand auf mehrere Rechner verteilen. Eine verteilte Ausführung ist beispielsweise immer dann notwendig, wenn Algorithmen zur Bewegungserzeugung oder Sensordatenverarbeitung so zeitaufwendig sind, dass die Realisierung der notwendigen Regelungsfrequenz mit einem einzigen Rechner nicht möglich ist.

Zur Veranschaulichung des auf der Middleware XD basierenden Verteilungsmechanismus von Steuerungskomponenten wird die Verteilung eines Bewegungsmoduls (Motion Module: MM) anhand eines Aktivitätsdiagramms in Abbildung 6-9 dargestellt. Die Abbildung zeigt den Master-Rechner, auf dem die Hauptsteuerungskomponenten (nicht in Abbildung dargestellt) ausgeführt werden und einen Remote-Rechner, auf dem das verteilt ausgeführte Bewegungsmodul (Remote Motion Module) abgebildet ist.

Damit die Steuerung eine verteilte Ausführung unterstützen kann, wird sie um ein Proxy-Modul erweitert. Der Proxy übernimmt sämtliche Datenübermittlungen zwischen der Kommunikation mit dem lokalen und dem verteilten Bewegungsmodul während des ganzen verteilten Rechenprozesses. Weiterhin sorgt der Proxy bei der Initialisierung dafür, dass das anzuwendende Verteilungsmuster aus einer xml-basierenden Konfigurationsdatei ausgelesen und Steuerungsmodule (in diesem Kontext auch Task genannt) ermittelt werden, die nicht lokal, sondern verteilt auf einem Remote-Rechner ausgeführt werden sollen. Anschließend setzt der Proxy entsprechende globale Flags, anhand derer die einzelnen Tasks erkennen, ob sie lokal oder verteilt auf einem Remote-Rechner ausgeführt werden sollen. Die Synchronisation zwischen dem Proxy und dem zu verteilenden Bewegungsmodul erfolgt über eine Condition Variable, die die Zustände „SendData“ und „DataReceive“ annehmen kann.

Im zyklischen Betrieb erfolgt die verteilte Ausführung folgendermaßen: das Bewegungsmodul wird lokal vom Steuerungskern via Nachrichtenversand zur Ausführung eines bestimmten Bewegungsalgorithmus aufgefordert. Anhand des vom Proxy gesetzten Flags erkennt das lokale Bewegungsmodul, dass der Bewegungsalgorithmus verteilt ausgeführt werden soll. Anstatt den Algorithmus nach Erhalt der Nachricht selbst auszuführen, kopiert das Bewegungsmodul die per Nachricht übermittelten Daten in einen mit dem Proxy vorher vereinbarten Shared-Memory-Bereich und signalisiert den blockierten Proxy durch Setzen des Synchronisationsstatus „SendData“ in der Condition Variable. Der Proxy liest anschließend die Daten aus dem Shared-Memory-Bereich aus uns erstellt daraus eine synchrone REQUEST-Nachricht, die über die Middleware XD nach dem im Abschnitt 6.8 beschriebenen Ansatz an den Remote-Rechner übertragen wird. Danach blockiert der Proxy wieder, bis er eine Reply-Nachricht vom Remote-Modul erhält. Auf dem Remote-Rechner reagiert das Modul auf die Nachricht und entnimmt ihr die Bewegungsdaten auf denen er den Bewegungsalgorithmus ausführt. Das Berechnungsergebnis des Bewegungsalgorithmus wird anschließend in einer REPLY-Nachricht

kopiert und über XD an den Master-Rechner geschickt. Dort entnimmt der Proxy die Berechnungsergebnisse aus der Nachricht und übermittelt sie über Shared-Memory an den lokalen Bewegungsmodul. Anschließend signalisiert er den lokalen Bewegungsmodul durch das Setzen des Zustands „DataReceived“ in die Condition Variable. Auf diese Weise ist das lokale Bewegungsmodul in der Lage, das Ergebnis des Bewegungsalgorithmus, der verteilt auf einem Remote-Rechner ausgeführt wurde, an den Steuerungskern zu übermitteln.

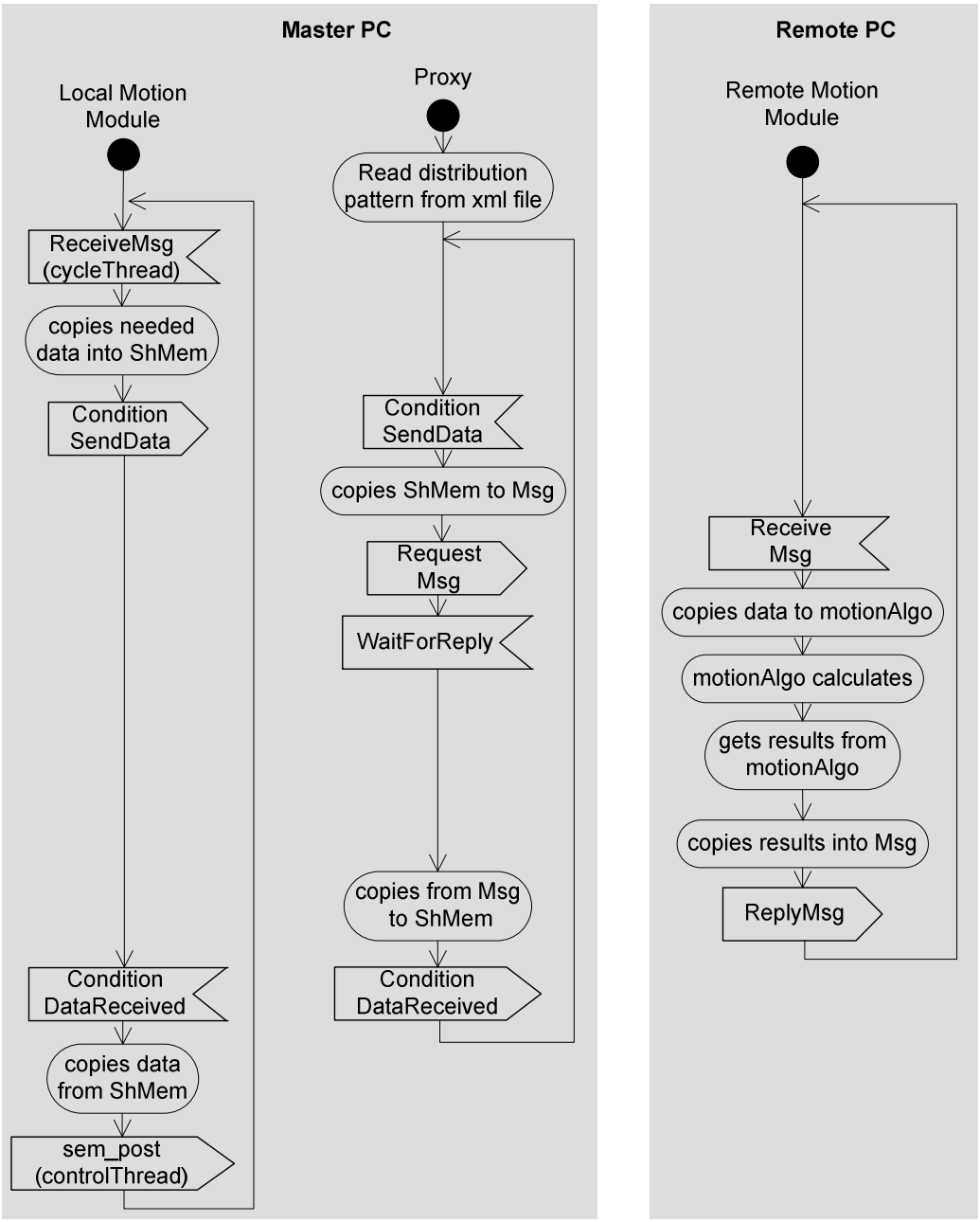

Abbildung 6-9: Aktivitätsdiagramm zur Veranschaulichung des angewendeten Mechanismus zur Verteilung eines Bewegungsmoduls

6.13 Zusammenfassung

In diesem Kapitel wurde die Entwicklung und Implementierung der verteilten Middleware XD beschrieben. XD wird im PC-basierten Steuerungskonzept des Sonderforschungsbereich SFB562 eingesetzt, um die Berechnung aufwendiger Algorithmen im zyklischen Betrieb auf verteilten Rechnern auszuführen.

Mit XD lässt sich ein Master/Slave Verteilungsmodell realisieren. Dabei fungiert der bisherige Hauptsteuerungsrechner als Master. Neben der durch das IAP geregelten zyklischen Kommunikation, die zwischen dem Master und den Sensor- und Aktor-Einheiten des Roboters stattfindet, wurde aus Echtzeit- und Performanzgründen eine zweite physikalische Busverbindung zwischen Master und den Slaves realisiert. Im Bezug auf die erreichbare Performanz hat eine Untersuchung ergeben, dass der IEEE1394 Standard das optimale Bussystem für diese Verbindung ist.

Um die Skalierbarkeit des verteilten Systems zu gewährleisten, unterstützt XD die dynamische Integration neuer Slave-Rechner im laufenden Betrieb. Dafür sind Mechanismen in XD integriert, die eine automatische Registrierung von entfernten Ressourcen auf dem zentralen Master-Rechner realisieren. Weiter wurde die Registrierungsroutine von XD so erweitert, dass jede Ressourcenregistrierung auf einem XD-Slave automatisch an den XD-Master übermittelt wird. Um nach dem Beenden eines XD-Slaves Funktionsfehler im verteilten System zu vermeiden und die Systemzuverlässigkeit zu gewährleisten, wurde ein Sperrmechanismus in den XD-Slave integriert, der das Beenden nur dann zulässt, wenn sämtliche zuvor registrierte Ressourcen abgemeldet wurden.

Unter XD erfolgt der Zugriff auf entfernte Ressourcen über dem auf Netzwerkebene ausgeweiteten *Message-Passing* Mechanismus. Dabei macht XD den Einsatz der bisherigen nachrichtenbasierten COMMAND- und REQUEST-Mechanismen im verteilten System für den Anwender transparent. Die bei bidirektionaler Kommunikation erzielbare Performanz erfüllt die für SFB562-Steuerungsapplikationen formulierten Echtzeitanforderungen.

Im nächsten Kapitel wird, im Anschluss an die Beschreibung des für die Integration von Self-X-Eigenschaften in die Steuerung entwickelte System-Monitorings, der Entwurf und die Realisierung einer durch die Middleware XD und einen entwickelten Self-Manager gestützten dynamischen Verteilung der Robotersteuerungskomponente vorgestellt.

7 System-Monitoring

Über die Möglichkeit des Zugriffs auf verteilte Software-Komponenten der Steuerung können algorithmisch rechenaufwendige Verfahren (z.B. Singularität vermeidende Bahnplaneralgorithmen, Sichtsysteme und bildverarbeitende Algorithmen) in den deterministischen Steuerungszyklus integriert werden. Allerdings kann eine suboptimale Verteilung der Steuerungskomponenten die gesamte Kommunikationslatenz erhöhen und im schlimmsten Fall sogar zur Verletzung des Timings des Steuerungszyklus führen. Die große Menge an Verteilungsmustern (Paarungen verfügbarer CPU-Ressourcen und zu verteilende Steuerungskomponenten) wirft außerdem die Frage auf, welches dieser Verteilungsmuster für verschiedene Situationen jeweils am besten geeignet ist [SGM09].

Um die Verwaltungskomplexität in den Griff zu bekommen und eine automatische zeitoptimale Nutzung der im verteilten System verfügbaren Rechenressourcen zu erreichen, wurde im SFB562 unter anderem als Ziel gesetzt, eine Managementkomponente [MHS+07] [MSA+08] [SH07] [SHG07] in die Steuerungssoftware zu integrieren. Die Managementkomponente ist an die von IBM im Autonomic-Computing-Kontext vorgeschlagenen Autonomic-Manager eng angelehnt [KC03]. Sie soll selbständig und zeiteffizient auf dynamische Änderungen der Verfügbarkeit von Rechenressourcen im verteilten System sowie auf sich ändernde strategische Anforderungen an die Steuerung reagieren und eine automatische Laufzeitanpassung der Verteilung der Steuerungsmodule ausführen können. Ferner dürfte die Integration der Managementkomponente in die Steuerung deren Echtzeitverhalten nicht beeinträchtigen.

In diesem Kapitel werden die für die Integration einer Managementkomponente in die Steuerungssoftware notwendigen Mechanismen behandelt, mit denen die Kommunikations-Infrastruktur ergänzt wurde. Insbesondere wird der in der Middleware integrierte System-Monitor beschrieben, der topologische und zeitliche Informationen über die Ausführung von Software-Tasks im verteilten System zur Laufzeit ermittelt und der Managementkomponente bereitstellt. Anschließend wird anhand eines Applikationsbeispiels der Einsatz von Netzwerkmanagementkomponenten in ein verteiltes Robotersteuerungssystem veranschaulicht.

7.1 Grundlage des autonomen Computings

Das Konzept vom autonomen Computing wurde erstmals 2001 von IBM in einem Manifest eingeführt [KC03]. Dort wurde das Konzept wie folgt definiert: „Autonomes Computing ist ein Ansatz, selbst verwaltete Computersysteme zu erreichen, die ein Minimum an menschlichem Eingreifen erfordern“. Das ultimative Ziel des autonomen Computing ist es, den mit der Verwaltung und Steuerung von Systemen verbundenen Aufwand in die Hard- und/oder Software-System-Infrastruktur zu verschieben.

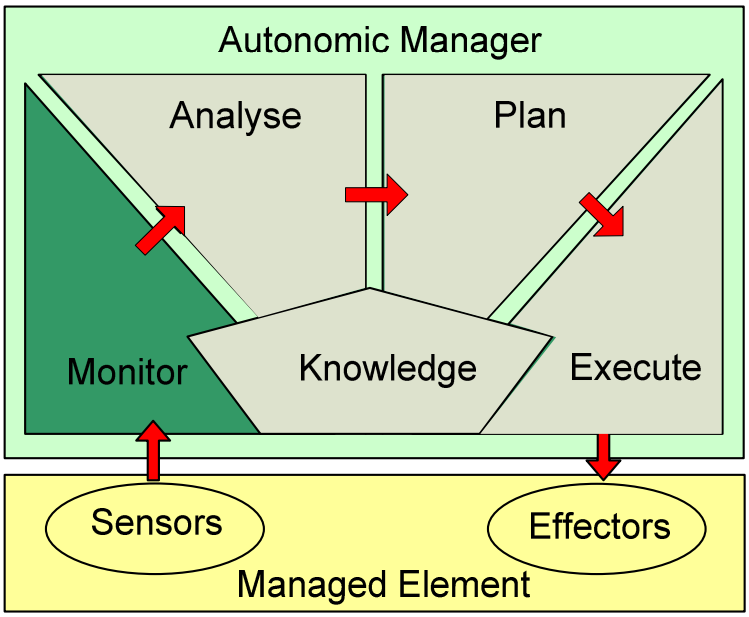

Abbildung 7-1: Architektur des autonomen Computings (nach [Ver09])

Gemäß der Architektur des autonomen Computing (Abbildung 7-1) kann jedes sich selbst verwaltende System in einen autonomen Manager und ein verwaltetes Element aufgeteilt werden. Das verwaltete Element besteht aus Sensoren, die Informationen über den aktuellen Zustand des Elements liefern, die vom Manager gelesen werden können, und Effektoren, die als Schnittstelle fungieren, über die das verwaltete Element angesprochen und beeinflusst werden kann. Der autonome Manager ist für die Verwaltung des autonomen Elements zuständig. Er enthält Algorithmen, die dem System ihre Self-X-Eigenschaften (Selbst-Konfiguration, Selbst-Optimierung, Selbst-Heilung und Selbst-Schutz) verleihen.

Der autonome Manager selbst besteht aus vier verschiedenen funktionalen Komponenten: eine Monitor-Komponente, eine Analyse-Komponente, Planungs-Komponente und eine Ausführungs-Komponente. Die Monitor-Komponente ist für die Erfassung von Status und Verhaltensinformationen bezogen auf das verwaltete Element zuständig. Anhand der erfassten Daten kann die Analyse-Komponente herausfinden, ob das System ein Fehlverhalten aufweist. Während die Planungs-Komponente dafür verantwortlich ist, die Korrekturmaßnahmen zu bestimmen, die den Systemfehler beheben sollen, sorgt die Ausführungs-Komponente über ein geeignetes Interface für eine einwandfreie Durchführung dieser Maßnahmen. Zur Bestimmung der richtigen Korrekturmaßnahme kann die Planungs-Komponente eine Vielfalt von Algorithmen anwenden. Ein einfacher Ansatz besteht darin, eine Datenbank mit Expertenwissen (Knowledge) vorzuhalten, auf der jedem Systemfehler eine entsprechende Korrekturmaßnahme zugeordnet wird [Ver09].

Bei der Integration von autonomen Computing-Eigenschaften in den RCA562-Kontext wird die gesamte Steuerungssoftware als das zu verwaltende Element betrachtet, das mit Hilfe eines Self-Managers in ein Self-X-fähiges System verwandelt werden soll. Da die Steuerung modular

aufgebaut ist und alle Softwaremodule über die von der Middleware MiRPA-XD bereitgestellten nachrichtenbasierten Mechanismen kommunizieren, lässt sich das Softwareverhalten durch die Auswertung des Nachrichtenverkehrs innerhalb der Middleware beobachten; auf diese Weise wird die für das autonome Computing notwendige Monitor-Komponente aufgebaut. Zudem lässt sich die Performanz von einzelnen Steuerungs-Tasks über die Integration von Messpunkten in das nachrichtenbasierte Kommunikationsinterface quasi „on the fly“ ermitteln.

7.2 Aktives und passives Monitoring

Das Monitoring eines Computersystems lässt sich allgemein als ein Prozess der Erfassung und Darstellung von applikationsrelevanten Systeminformationen beschreiben. Die Systeminformationen umfassen die Status- und Konfigurationsinformationen sowie die Performanzdaten und die Topologieinformationen. Die Performanzinformationen schließen die Laufzeiten und Latenzen von Anfragen ein, die innerhalb von Systemkomponenten abgearbeitet werden.

In echtzeitfähigen Systemen besteht die Herausforderung des Monitorings darin, dass die Monitoring-Aktivitäten keine oder nur minimal negative Auswirkungen auf die regulären Betriebsaktivitäten haben dürfen. Insbesondere dürfte das Monitoring keine Auswirkung auf die Ausführung von Anwenderapplikationen haben. Um dieser Vorgabe gerecht zu werden, bietet es sich an, einen passiven Monitoring-Ansatz zu realisieren. Beim passiven Monitoring erfolgt die Erfassung und Übermittlung von Systeminformationen an die höheren Schichten des Self-Managers automatisch und ohne explizite Aufforderung seitens des Anwenders. Die Datenerfassung und -übermittlung nimmt keine zusätzlichen Ressourcen in Anspruch. Für die Übermittlung der erfassten Daten ist kein gesonderter Nachrichtenverkehr notwendig; die Daten werden an Nachrichten angehängt, die im regulären Verkehr gesendet werden. Ein Nachteil dieses Ansatzes ist die eingeschränkte Flexibilität der Datenübermittlung, die von der Dichte des regulären Verkehrs abhängig ist. Im Gegensatz zum passiven Monitoring erfolgt die Übermittlung der erfassten Daten beim aktiven Monitoring mittels eines zusätzlich initiierten Datentransfers. Auf diese Weise ist die Datenübermittlung beim passiven Monitoring zwar flexibler, aber sie erhöht die CPU-Last und beeinflusst gleichzeitig die Ausführung von Anwenderapplikationen.

7.3 Definition und Ziel

Der System-Monitor ist die Softwarekomponente, die in die MIRPA-XD Umgebung eingesetzt wird, um topologische und zeitliche Informationen über die Ausführung von Tasks zur Laufzeit zu gewinnen. Die topologischen Informationen beschreiben unter anderem die Netzwerkteilnehmer, das vernetzende Bussystem und die Task-Verteilung auf den einzelnen Netzwerkteilnehmern. Der System-Monitor wird außerdem zur Überwachung von Tasks eingesetzt, deren Ausführung eine vorgegebene zeitliche Grenze im Echtzeitkontext nicht

überschreiten darf. Bei der Ausführung des Task-Monitorings muss darauf geachtet werden, dass die Performanz des gesamten Systems nicht durch die Datengewinnung beeinträchtigt wird; aus diesem Grund wird ein passives Monitoring bevorzugt.

Aus Sicht des Anwenders ist ein Task eine in sich geschlossene, eindeutig identifizierbare Aufgabe. Ein Task beschreibt eine rechnende Softwarekomponente, die Eingabedaten auf Anfrage nach einem festgelegten Algorithmus (oder Schema) bearbeitet und Ergebnisse in einem vorgegebenen Format zurückliefert. Die Komplexität einer Aufgabe ist für die Task-Auslegung nicht relevant. In dem Steuerungskontext kann ein Task unter anderem ein Bahnplaner, ein Sensordaten aufbereitender Algorithmus, oder ein unterlagerter roboterspezifischer Regelungsalgorithmus sein.

Aus Sicht der Middleware ist ein Task ein Anwender Programm-Code, dessen Beginn und Ende unmittelbar durch zwei zusammenhängende Interface-Funktionen festgelegt sind. In diesem Kontext sind zusammenhängende Interface-Funktionen solche, die einen Kommunikationsvorgang vollständig beschreiben. Ein Kommunikationsvorgang unter dem Client/Server Modell wird zum Beispiel durch eine Anfrage und die dazugehörige Antwort beschrieben. In MIRPA-XD werden Tasks sowohl in Client- als auch in Server-Modulen eingekapselt; jedes Modul kann dabei mehrere Tasks beinhalten. Die Identifizierung eines Tasks erfolgt über dessen Namen, der im ganzen System eindeutig sein muss.

7.4 Architektur des System-Monitors

Es gibt zwei wesentliche Anforderungen, die der Monitor erfüllen muss. Zum einen muss er konfigurierbar sein: das hieß, der Anwender (in diesem Fall der Self-Manager) sollte in der Lage sein zu bestimmen, welche Tasks im laufenden Betrieb überwacht werden sollten. Zum anderen dürften die Monitoring-Aktivitäten keine Auswirkungen auf das Echtzeitverhalten des Systems haben. Ferner dürften sie die Systemperformanz nicht oder nur minimal beeinflussen. Orientiert an diesen Anforderungen wurde die in Abbildung 7-2 dargestellte Architektur für den System-Monitor konzipiert. Links im Bild befindet sich die Middleware MIRPA-XD mit der integrierten Monitoring-Erweiterung, in der Mitte ist das Monitoring-Interface zum Self-Manager und rechts exemplarisch die in die Steuerungssoftware zu integrierende Self-Manager-Applikation (Self-X) dargestellt.

Um den System-Monitor zu integrieren, wurde die Middleware um eine Monitoring-Komponente, bestehend aus einem nebenläufigen Thread namens „SystemMonitor“ und einem FIFO erweitert. Im laufenden Betrieb werden die Ausführungszeiten der registrierten Tasks in den Middleware-Komponenten *ObjectServer*, *Scheduler* and *Remote Control* aufgenommen (die Erfassung der Task-Ausführungszeiten wird in Abschnitt 7.5 erläutet). Um die Middleware-Aktivitäten nicht zu beeinträchtigen, werden die Daten an den System-Monitor-Thread zur Bearbeitung auf niedrigerer Priorität übergeben. Die Übergabe erfolgt über einen Eintrag (Push) in das Monitor-FIFO und eine anschließende Aktivierung des System-Monitor-Threads. Nach

Aktivierung bearbeitet der System-Monitor-Thread sämtliche FIFO-Einträge: er berechnet die effektive Ausführungszeit (siehe Abschnitt 7.7) jedes Tasks, trägt sie in das Monitor-Dateninterface ein und löst einen Alarm aus, falls erforderlich. Auf diesen Alarm reagiert anschließend die Self-Manager-Applikation.

Das Monitor-Interface ist die Schnittstelle zwischen dem Monitor und der Self-Manager-Applikation. Sie besteht aus einer XML-basierten Konfigurationsdatei, einem Dateninterface und einer Signalisierungseinheit. In der Konfigurationsdatei werden offline die Tasks spezifiziert, die überwacht werden sollen. Zusätzlich können Zeitgrenzen für jeden Task spezifiziert werden, bei deren Überschreitung der Monitor einen Alarm auslösen soll. Außerdem kann die anfängliche Systemtopologie und Task-Verteilung spezifiziert werden. Das Dateninterface ist das Medium, worüber Daten zwischen dem Monitor und dem Self-Manager ausgetauscht werden. Es ist als Shared-Memory realisiert. Während der Hochlaufphase der Middleware wird es mit Daten aus der XML-Datei initialisiert. Im laufenden Betrieb wird das Dateninterface auf der einen Seite mit den ermittelten Task-Ausführungszeiten aktualisiert, auf der anderen Seite von dem Self-Manager ausgelesen.

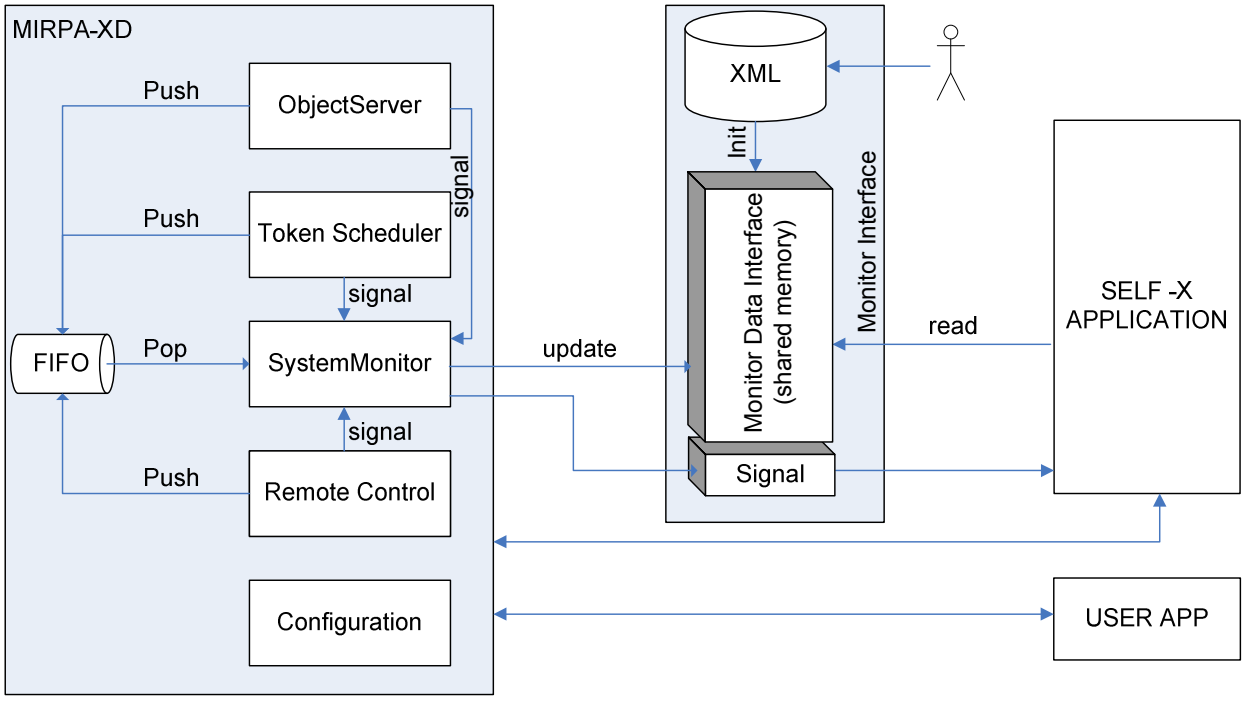

Abbildung 7-2: Architektur des System-Monitors

7.5 Task-Monitoring im MIRPA-XD Kontext

Da die Tasks in den Middleware-Interface-Funktionen eingebettet sind, lässt sich mit wenig Aufwand deren Ausführungszeit, durch die Integration von Messpunkten, im laufenden Betrieb

ermitteln. Abhängig von dem gewählten Kommunikationsmechanismus kann man unter MIRPA-XD zwischen folgenden Typen von Tasks unterscheiden:

- REQUEST-Task
- COMMAND-Task
- TOKEN-Task
- Aktiver User-Task

7.5.1 REQUEST-Task

Ein REQUEST-Task lässt sich mit Hilfe des synchronen Request/Reply Kommunikationsmechanismus in Server-Applikationen realisieren. Der systemweit eindeutige Name einer Request-Nachricht fungiert hierbei als Name des Tasks. Im ganzen System wird der Task durch eine registrierte Request-Nachricht vollständig beschrieben. Abbildung 7-3 stellt die Einbettung eines REQUEST-Tasks in den Request/Reply Mechanismus sowie die automatische Erfassung der Ausführungszeit in einem Sequenzdiagramm dar. Das Diagramm besteht aus drei Prozessen:

- der Middleware MIRPA-XD bestehend aus dem Monitor- und dem ObjectServer-Thread
- einer Server-Applikation, die den Task implementiert und aus einem Rx- und Main-Thread besteht
- der Self-Manager-Applikation

Außerdem werden das Monitor-FIFO und das Monitor-Dateninterface zum Self-Manger dargestellt. In der Ausgangssituation wartet die Server-Applikation über eine API-Funktion (*MIRPA_ReceiveMsg*) blockiert auf Anfragen. Nach Erhalt einer vom Client initiierten und vom ObjectServer-Thread weitergeleiteten Request-Nachricht führt sie den zugehörigen Algorithmus aus und sendet die Ergebnisse über eine REPLY-Nachricht zurück an den Client. Der Anwender-Task ist also in diesem Fall zwischen den API-Funktionen *MIRPA_ReceiveMsg()* und *MIRPA_ReplyMsg()* eingebettet. Dies ermöglicht eine automatische Erfassung der Task-Ausführungszeit ohne Modifikation des Anwender-Programmcodes. Durch eine Integration von Messpunkten (T1, T2 in Abbildung 7-3), einerseits in *MIRPA_ReceiveMsg()* unmittelbar nach Erhalt der REQUEST-Nachricht und anderseits in *MIRPA_ReplyMsg()* unmittelbar vor dem Zurücksenden der Ergebnisse, wird die Task-Ausführungszeit im laufenden Betrieb erfasst. Die erfasste Ausführungszeit wird in die REPLY-Nachricht integriert und an den ObjectServer mitgeschickt. Der ObjectServer trägt anschließend die Ausführungszeit und eine entsprechende Task-Referenz in das FIFO ein und aktiviert anschließend den Monitor-Thread. Letzterer holt die Daten aus dem FIFO, aktualisiert das Monitor-Dateninterface, und benachrichtigt den Self-Manager, der entsprechende Maßnahmen treffen kann.

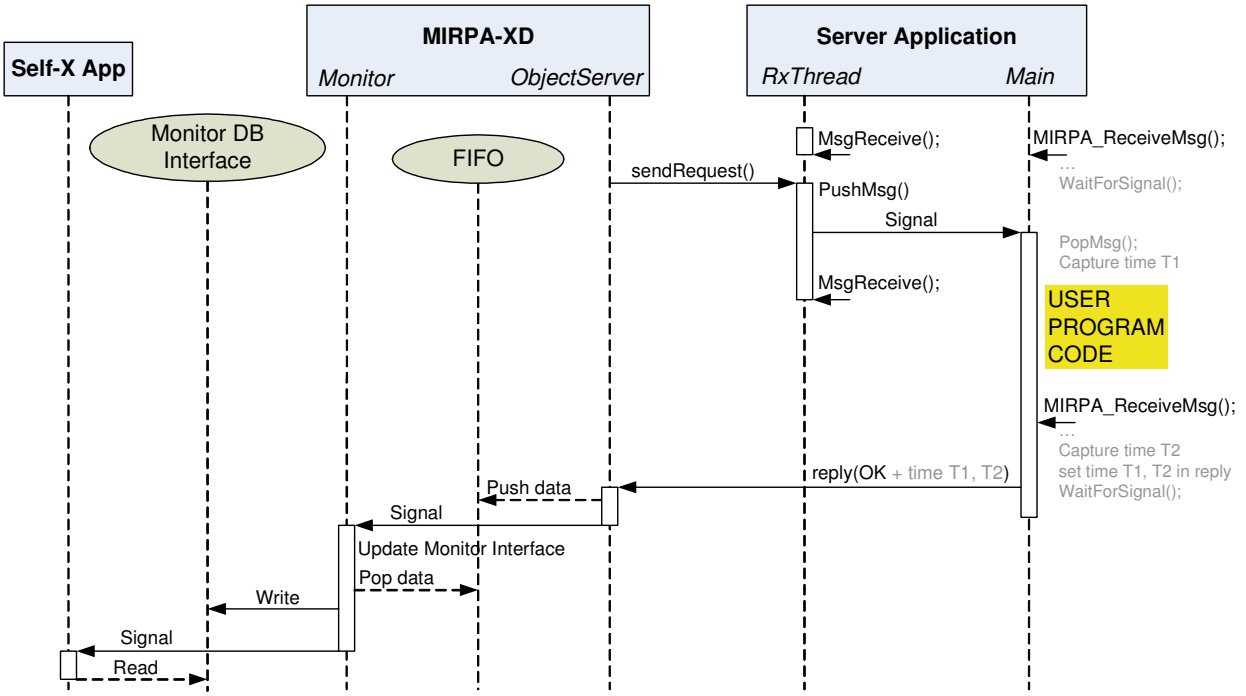

Abbildung 7-3: Einbettung eines Tasks in den synchronen Request/Reply Kommunikationsmechanismus und automatische Erfassung der Ausführungszeit im laufenden Betrieb.

7.5.2 COMMAND-Task

Ein COMMAND-Task lässt sich mit Hilfe des asynchronen COMMAND-Kommunikationsmechanismus in Server-Applikationen realisieren. Auch in diesem Fall fungiert der systemweit eindeutige Name einer Command-Nachricht als Name des Tasks. Eine Command-Nachricht ermöglicht den Aufbau eines asynchronen Kommunikationskanals zwischen einem Client und einem Server. Der Client wartet in diesem Falls nicht blockierend auf die Antwort des Servers und ist von dem Server zeitlich entkoppelt. Da der Server in diesem Fall keine Antwort-Nachricht an den Client versendet, wird die Einbettung eines Tasks und die automatische Erfassung ihrer Abarbeitungszeit erschwert. Zudem können die erfassten Daten nicht ohne zusätzlichen Kommunikationsaufwand dem Monitor mitgeteilt werden. Dies kann allerdings unter bestimmten Umständen die Performanz des gesamten Systems beeinträchtigen.

Da die Task-Ausführungszeit automatisch erfasst werden soll, muss sie zwischen zwei API-Funktionen der Middleware eingebettet werden. Im Fall des COMMAND-Mechanismus steht nur die API-Funktion *MIRPA_ReceiveMsg()* zur Verfügung. Deshalb werden Tasks zwischen zwei aufeinander folgenden *MIRPA_ReceiveMsg()* eingebettet. Die automatische Erfassung der Task-Ausführungszeit erfolgt durch die Integration von Messpunkten in *MIRPA_ReceiveMsg()*. Das in Abbildung 7-4 dargestellte Sequenzdiagramm veranschaulicht diese Zeiterfassung. Der Startpunkt der Messung (capture time T1) liegt in *MIRPA_ReceiveMsg()* nach dem Erhalt der COMMAND-Nachricht und unmittelbar bevor der Anwender-Code ausgeführt wird. Der

Endpunkt der Zeitmessung (capture time T2) ist in dem nächsten Aufruf von *MIRPA_ReceiveMsg()* integriert. Selbstverständlich ist die Erfassung der Task-Ausführungszeit auf diese Weise nur dann möglich, wenn MIRPA_ReceiveMsg() erneut unmittelbar nach der Task-Ausführung aufgerufen wird.

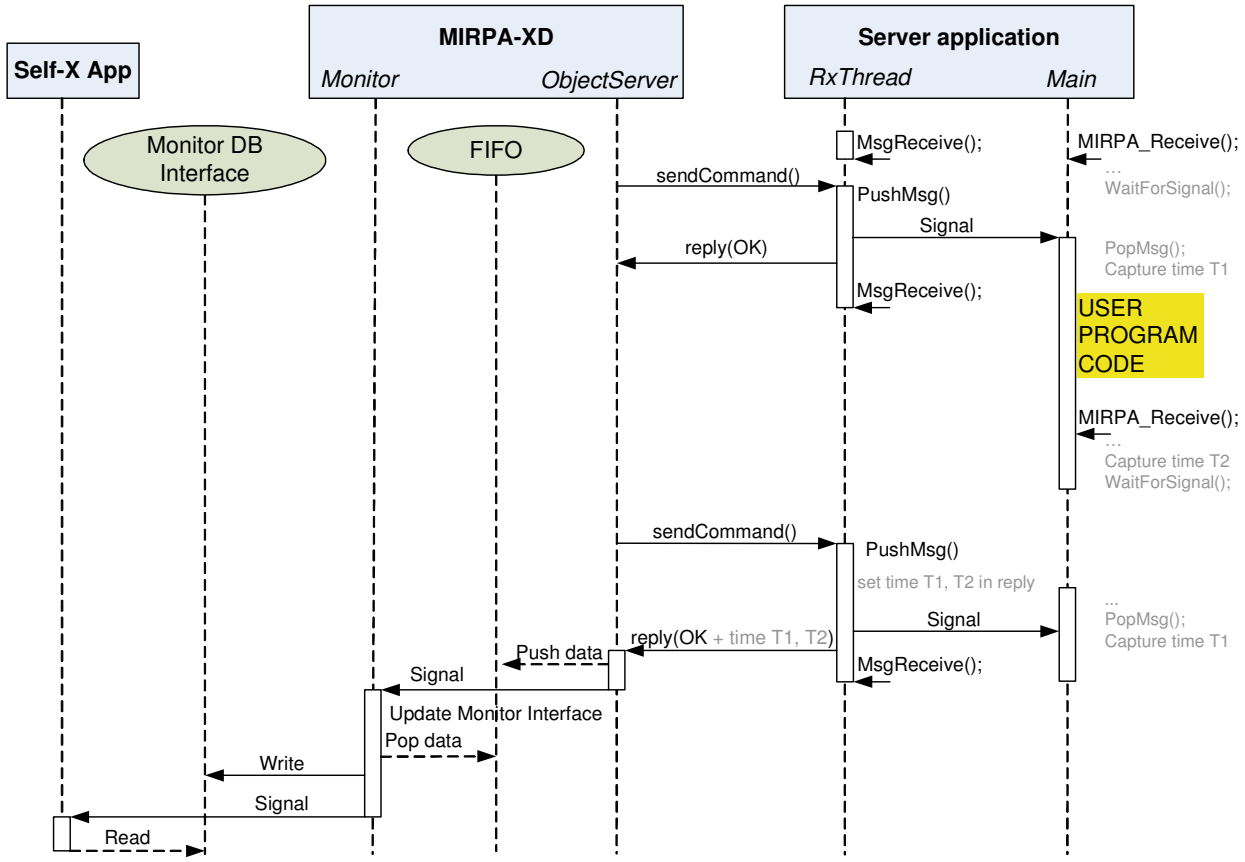

Abbildung 7-4: Einbettung eines Tasks in den asynchronen COMMAND Kommunikationsmechanismus und automatische Erfassung der Ausführungszeit im laufenden Betrieb.

Die erfassten Daten können dem Monitor auf verschiedenen Wegen mitgeteilt werden. Im Folgenden werden drei Ansätze dargestellt.

Übermittlung über Shared-Memory

Bei diesem Ansatz werden die gemessenen Daten in einen dafür vorgesehenen Shared-Memory-Bereich geschrieben. Anschließend wird der Monitor durch ein Signal aktiviert. Daraufhin durchsucht der Monitor-Thread sämtliche Shared-Memory Regionen und extrahiert die entsprechende Zeitinformation. Ein Nachteil dieses Ansatzes ist der zusätzliche Verwaltungsaufwand und Ressourcenverbrauch seitens der Middleware. Denn für jede registrierte COMMAND muss eine zugeordnete Shared-Memory für das Monitoring registriert und verwaltet werden. Außerdem unterstützt dieser Ansatz nicht die automatische Erfassung der

Zeitinformation von Tasks, die über Rechnergrenzen hinaus verteilt sind. Der Grund dafür ist, dass die Shared-Memory-Kommunikation nicht über Rechnergrenzen unterstützt wird. Dieser Ansatz wurde deshalb nicht umgesetzt.

Übermittlung über Message-Passing

Im Gegensatz zu dem Shared-Memory Ansatz wird die Task-Ausführungszeit in diesem Fall über eine Nachricht an den Monitor gesendet. Dank des Message-Passing, das auch im verteilten System unterstützt wird, kann die Ausführungszeit von Tasks, die über Rechnergrenzen hinweg verteilt sind, automatisch erfasst und an den Monitor übermittelt werden. Allerdings verursacht dieses aktive Monitoring zusätzliche unerwünschte Performanzeinbuße im System und wurde deshalb auch nicht umgesetzt.

Passive und zeitversetzte Übermittlung

Dieser letzte Ansatz hat den Vorteil, dass die Daten ohne zusätzlichen Kommunikationsaufwand an den Monitor übermittelt werden. Die asynchrone Kommunikation wird durch den von QNX-Neutrino bereitgestellten synchronen Message-Passing-Mechanismus realisiert. Um eine zeitliche Entkopplung zwischen dem Client und dem Server zu erreichen, quittiert der Server jede ankommende COMMAND-Nachricht mit einem OK (Abbildung 7-4, reply(OK)), ehe er die dazugehörige Anfrage bearbeitet. Die Quittierung wird in diesem Ansatz benutzt, um die Task-Ausführungszeit an den Monitor zu übertragen; dadurch erübrigt sich die Notwendigkeit, aktiv eine gesonderte Nachricht für die Datenübermittlung einzusetzen. Dazu wird die Quittierungs-Nachricht um die Messpunkte (T1, T2) als zusätzlicher Parameter erweitert. Die gemessene Ausführungszeit wird zeitversetzt, erst mit der nächsten Anfrage, an den Monitor übertragen. Dieser Ansatz verursacht einen geringen Verwaltungsaufwand und keinen zusätzlichen Kommunikationsaufwand. Er bietet deshalb die grundlegende Voraussetzung für die Integration von Monitoring-Funktionen ohne oder mit geringem Einfluss auf die gesamte Systemperformanz.

7.5.3 TOKEN-Task

Der TOKEN-Task lässt sich mit Hilfe des synchronen Token-basierten Kommunikations-mechanismus realisieren; hierbei werden Tasks in Token-Funktionen eingebettet. Das Sequenzdiagramm in Abbildung 7-5 zeigt die Einbettung von Tasks in Token-Funktionen. Neben dem Monitor-Thread wird der Scheduler-Thread in dem MIRPA-XD Prozessblock dargestellt. Außerdem werden exemplarisch zwei Anwender Token-Threads namens Token1 und Token2 dargestellt. Die Token-basierte Kommunikation erfolgt auf höchster Prioritätsebene und wird vom Scheduler zyklisch aktiviert. Um an der Token-basierten Kommunikation teilzunehmen, muss jede Anwenderapplikation eine Funktion anmelden, die in dem Token-Thread ausgeführt werden soll: dies ist die Token-Funktion. Der Funktionsname, der bei der Anmeldung der Token-Funktion angegeben wird, fungiert als Name des Tasks und identifiziert

sie eindeutig im ganzen System. Innerhalb der Token-Funktion werden die API-Funktionen *MIRPA_WaitForToken()* und *MIRPA_ReleaseToken()* in einer While-Schleife aufgerufen.

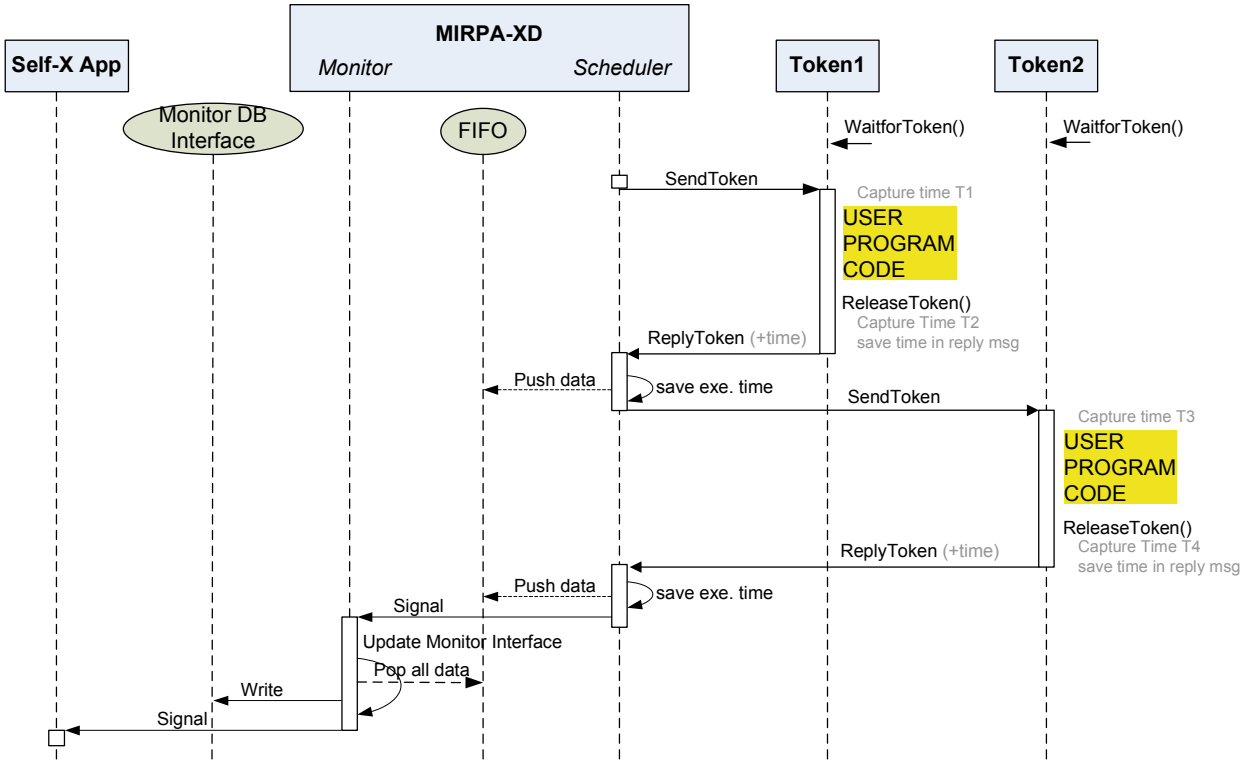

Abbildung 7-5: Einbettung von Tasks in Token-Funktionen, automatische Erfassung der Task-Ausführungszeit im laufenden Betrieb und Datenübermittlung an den Monitor

Im zyklischen Betrieb warten alle Token-Threads beim Aufruf von *MIRPA_WaitForToken()* blockierend auf den Token. Der Scheduler aktiviert die Token-Threads mittels des Versands eines Tokens. Nachdem der Scheduler einen Token-Thread aktiviert hat, blockiert er, bis er den Token zurück erhält. Nach Erhalt des Tokens wird der User-Programm-Code (Task) ausgeführt. Nach Beendigung der Task-Ausführung wird der Token durch die API-Funktion *MIRPA_ReleaseToken()* freigegeben. Der Task ist folglich innerhalb der API-Funktionen eingebettet und ihre Ausführungszeit kann automatisch erfasst und an den Monitor übermittelt werden. Die Ermittlung der Ausführungszeit erfolgt über die Integration von Messpunkten in die Funktionen *MIRPA_WaitForToken()* und *MIRPA_ReleaseToken()* (siehe Abbildung 7-5: T1, T2 und T3, T4). Die ermittelte Ausführungszeit wird bei der Freigabe des Tokens in die Token-Reply-Nachricht integriert und an den MiRPA-X-Scheduler geschickt, der anschließend die Ausführungszeit und eine entsprechende Task-Referenz in das Monitor-FIFO einträgt. Am Ende eines Token-Zyklus, in dem sämtliche Tasks aktiviert wurden, sendet der MiRPA-X-Scheduler ein Signal an den Monitor. Daraufhin holt der Monitor die Daten aus dem FIFO, aktualisiert das Monitor-Dateninterface, und signalisiert den Self-Manager, der entsprechende Maßnahmen treffen kann.

7.5.4 Aktiver User-Task

Der aktive User-Task ist nicht mit einem Middleware-Kommunikationsmechanismus gekoppelt. Dies gibt dem Anwender die Möglichkeit, einen beliebigen Block im Programmcode als Task zu definieren, die anschließend von dem Monitor überwacht werden kann. Die Notwendigkeit eines solchen Tasks ist durch die Komplexität des selbstoptimierenden Steuerungssystems gegeben, wo die Überwachung der Ausführungszeit von Komponenten gewährleistet werden muss, die nicht in Kommunikationsmechanismen der Middleware MIRPA-XD eingebettet sind. In Abbildung 7-6 wird die Vorgehensweise in einem Sequenzdiagramm exemplarisch erläutert. Eine Client-Applikation möchte die Ausführungszeit eines Programmcodes überwachen lassen. Dafür muss sie vier für das Monitoring bereitgestellte API-Funktionen aufrufen:

*MonitorRegisterTask(char *TaskName)*: mit dieser Funktion wird der Task bei der Middleware registriert. Der Name *TaskName* identifiziert den Task eindeutig im System. Die Funktion gibt eine Fehlermeldung zurück, falls der Name bereits im System vorhanden ist. Bei einer erfolgreichen Anmeldung wird ein Handle zurückgegeben, der für weitere Task-bezogenene Operationen eingesetzt werden muss. Mit Hilfe des Handle können mehrere User-Tasks innerhalb eines einzigen Prozesses definiert und überwacht werden.

MonitorStartTime(Handle): diese Funktion wird unmittelbar vor Beginn des Blocks im Programmcode gesetzt, der als User-Task festgelegt ist. Die Funktion legt den Start der Zeiterfassung fest.

MonitorStopTime(Handle): diese Funktion wird unmittelbar nach Ende des Blocks im Programmcode gesetzt, der als User-Task festgelegt ist. Sie legt das Ende der Zeiterfassung fest. Anschließend übermittelt sie die erfasste Task-Ausführungszeit über eine asynchrone Nachricht an den ObjectServer, der sie anschließend an den Monitor-Thread weiterleitet (Abbildung 7-6).

MonitorUnregisterTask(Handle): mit dieser Funktion wird der Task bei der Middleware abgemeldet und kann anschließend nicht mehr überwacht werden.

Da die Übermittlung der Task-Ausführungszeit an den Monitor über eine Nachricht erfolgt, findet in diesem Fall ein aktives Monitoring statt. Selbstverständlich kann dieser Vorgang im ungünstigsten Fall die Zustellung weiterer Nachrichten innerhalb der Middleware leicht verzögern und die Performanz der bereitgestellten Middleware-Kommunikationsmechanismen leicht beeinflussen. Die zeitliche Verzögerung liegt aber im Bereich unter einer Mikrosekunde und kann im Vergleich zu der Flexibilität bei der Task-Auslegung und –Überwachung vernachlässigt werden.

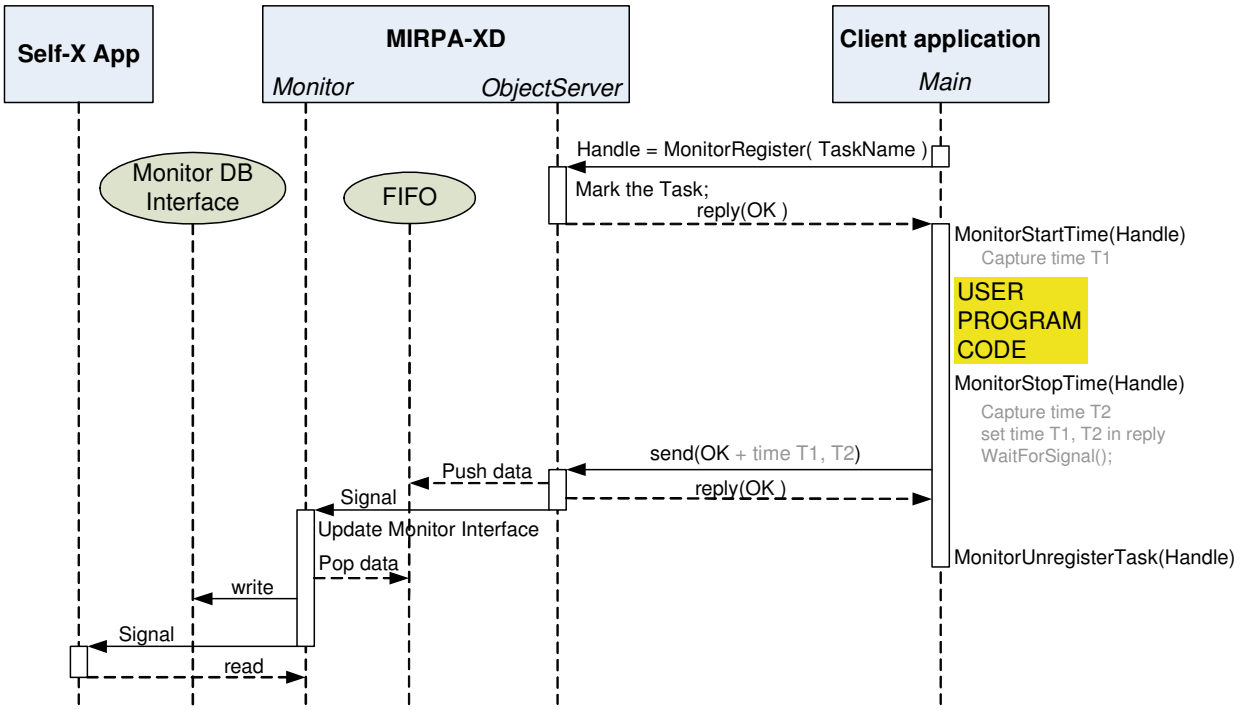

Abbildung 7-6: Realisierung des aktiven User-Tasks

7.6 Monitor-Datenverarbeitungsprozess

Nach dem Start der Middleware mit der entsprechenden Monitoring-Option wird die Ermittlung der Task-Ausführungszeiten automatisch aktiviert. Die gemessenen Zeiten werden aber nur dann vom System-Monitor berücksichtigt, wenn die dazugehörigen Tasks vorher vom Anwender zur Überwachung freigegeben wurden. Die Freigabe der zu überwachenden Tasks erfolgt anhand einer XML-basierenden Task-Konfigurationsdatei. Zur Beschreibung eines Tasks gehört unter anderem ein eindeutiger Name, eine Identifizierungsnummer (ID), der kleinste und der größte Wert der zur Laufzeit ermittelten Task-Ausführungszeit (BCET, WCET) [26]. Diese Beschreibungsdaten werden in einem entsprechenden Task-Datenobjekt im Monitor-Interface zusammengefasst (Abbildung 7-7) und dem Self-Manager zur Laufzeit zugänglich gemacht.

Unmittelbar nach dem Start liest die Middleware die Task-Konfigurationsdatei aus und initialisiert das Monitor-Interface mit den enthaltenen Task-Informationen. Bis auf die Informationen über die Task-Ausführungszeit (BCET und WCET), die erst zur Laufzeit ermittelt werden, werden die Task-Datenobjekte mit vom Self-Manager festgelegten Anfangswerten initialisiert.

Für eine schnelle Abarbeitung der Monitoring-Daten wird bei jeder Ressourcenregistrierung (Nachrichten oder Token-Funktionen) das entsprechende XD-Verwaltungsdatenobjekt indiziert.

[26] BCET: Best Case Execution Time, WCET: Worst Case Execution Time

Die Indizierung wirkt sich auf die Abarbeitung der Monitor-FIFO-Einträge so aus, dass sie einen direkten Zugriff auf das entsprechende Task-Datenobjekt ermöglicht und auf diese Weise eine zeitaufwendige Durchsuchung des Monitor-Interface zur Laufzeit erspart. Der gesamte Datenverarbeitungsprozess lässt sich, wie in Abbildung 7-7 dargestellt, in vier Schritte aufteilen.

- Erstens durchsucht die Middleware bei jeder Ressourcenregistrierung das zuvor initialisierte Monitor-Interface nach einem gleichnamigen Task.
- Zweitens trägt die Middleware bei erfolgreicher Suche den Index, der die Position des Task-Datenobjekts im Monitor-Interface beschreibt, in das XD-Verwaltungsdatenobjekt ein.
- Drittens trägt die Middleware, bei abgeschlossener Ermittlung der Task-Ausführungszeit, die ermittelten Task-Daten in das Monitor-FIFO ein. Neben den absoluten Werten für Start- und Endzeit der Task-Ausführung gehören ebenso der Task-Typ und der Task-Index zu dem einzutragenden FIFO-Element.
- Viertens erhält der Monitor-Thread während der Abarbeitung der Einträge des Monitor-FIFOs einen direkten Zugriff auf das im Monitor-Interface gespeicherte Task-Datenobjekt mit Hilfe des Task-Indexes.

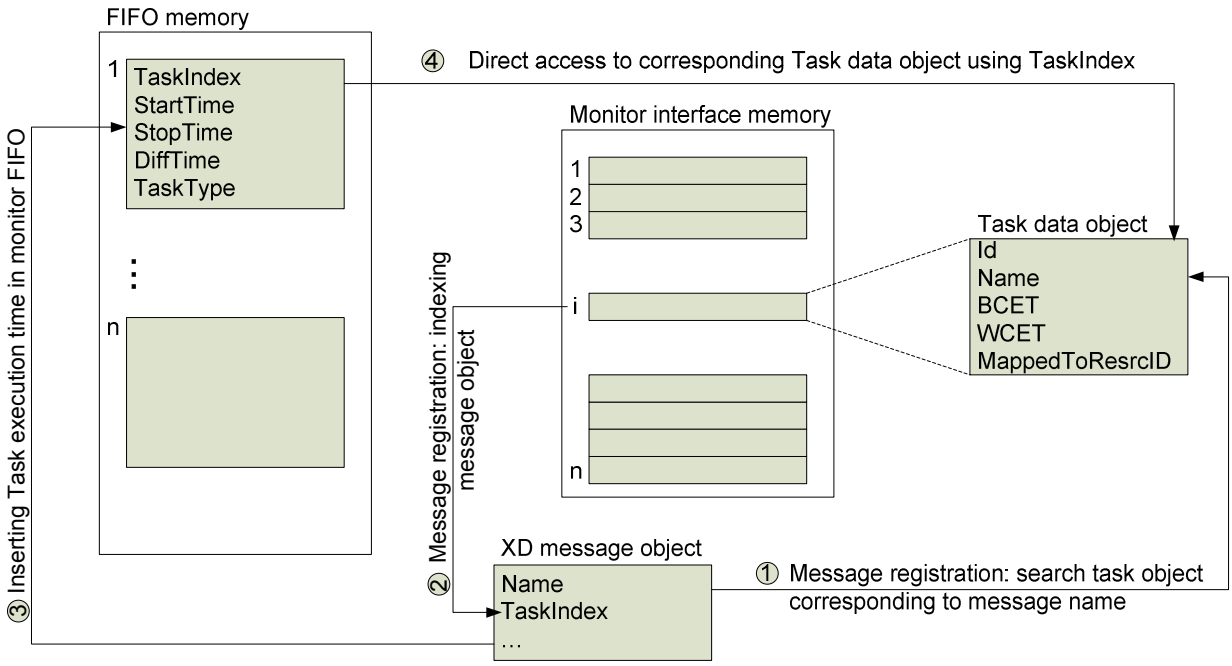

Abbildung 7-7: Verarbeitung der Monitoring-Daten in vier Schritten

7.7 Ermittlung der effektiven Task-Ausführungszeit

Die Ausführungszeit von Tasks wird durch die Integration von zwei Messpunkten jeweils am Anfang und am Ende eines Task-Programmcodes aufgenommen. Der mit einem Messpunkt

aufgenommene Zeitwert entspricht dem aktuellen Wert eines freilaufenden 64-Bit Zykluszählers. Dieser wird in jedem Prozessor implementiert, um Zeitmessungen mit hoher Auflösung (bis in Nanosekunden bei GHz Prozessoren) zu ermöglichen. Die Voraussetzung für eine fehlerfreie Messung der Task-Ausführungszeit ist, dass der Task ununterbrochen ausgeführt wird. Dies ist aber in Echtzeitsystemen mit prioritätsbasiertem Scheduling nicht gegeben, da ein Prozess mit hoher Priorität einen anderen Prozess mit niedriger Priorität unterbrechen kann. Die Unterbrechung führt dazu, dass die gemessene Zeit größer ist als die Zeit, in der der Task tatsächlich im Besitz der CPU ist. Die effektive Task-Ausführungszeit wird definiert als die Zeit, in der ein Task die CPU besitzt. Angenommen, ein Task kann bis zur vollständigen Ausführung mehrere Male unterbrochen werden, dann gilt folgende Gleichung:

$$T_{task} = T_{effektiv} + \sum_{i} T_{Unterbrechungi}$$

Die gemessene Task-Ausführungszeit ist demnach die Summe aus effektiver Task-Ausführungszeit und sämtlichen Unterbrechungszeiten. Um die effektive Zeit zu bestimmen, müssen von der gemessenen Zeit sämtliche Unterbrechungszeiten abgezogen werden. Nun muss geklärt werden, wie die Unterbrechungszeiten erfasst werden. Alle Prozesse, und damit auch alle Tasks in der MIRPA-XD Umgebung, werden mit einer Priorität ausgeführt, die höher ist als die höchste Priorität sämtlicher Betriebsystemkomponenten. Folglich kann ein Task nicht von einer Betriebssystemkomponente unterbrochen werden. Ein Task lässt sich also ausschließlich von anderen Tasks mit höheren Prioritäten unterbrechen. Demzufolge ist die für einen Task *A* zu berücksichtigende Unterbrechungszeit nichts anderes als die für einen anderen Task *B* gemessene Ausführungszeit.

Die Berechnung der effektiven Ausführungszeiten erfolgt über einen intervallbasierten Ansatz. Sämtliche gemessene Zeiten werden in den Monitor-FIFO eingetragen (siehe Abschnitt 7.4). Jeder Eintrag im FIFO entspricht einem Zeitintervall, das durch den Start- und den Endpunkt der Messung eines bestimmten Tasks festgelegt ist. Die Messzeiten eines Tasks werden erst in das Monitor-FIFO eingetragen, nachdem der Task vollständig ausgeführt wurde. Bei Unterbrechungen entspricht die Reihenfolge der FIFO-Einträge also nicht der chronologischen Reihenfolge der Task-Ausführung. In Abbildung 7-8 werden zur Veranschaulichung die FIFO-Einträge auf die Zeitachse projiziert. Aus dieser Projizierung kann beispielsweise erst abgelesen werden, dass die Ausführung eines Tasks i durch drei andere Tasks 1, 2, und 3 unterbrochen wurde, weil deren Zeitintervalle in dem Zeitintervall i liegen.

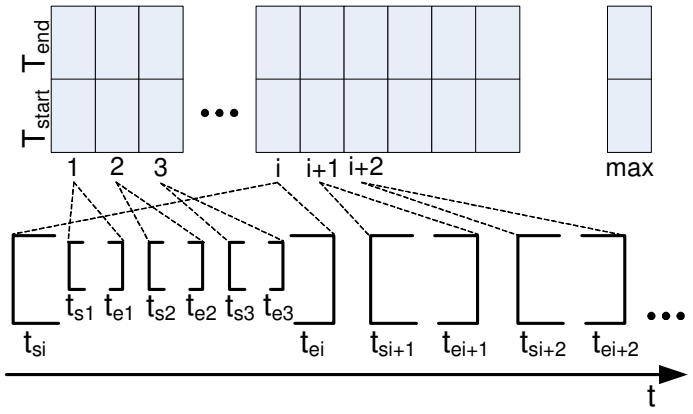

Abbildung 7-8: Projektion der zeitlichen Einträge aus dem Monitor-FIFO auf die Zeitachse

Die Ermittlung der effektiven Zeit für einen gegebenen Task i erfolgt nach dem folgenden Algorithmus:

1:.Initialisiere effektive Zeit mit Wert aus Differenz (t_{ei} - t_{si})
2:.**Für jeden** FIFO-Eintrag j **führe aus:**
3:…..**Wenn** Intervall j im Intervall i liegt, **dann**
4:…… Ziehe Länge von Intervall j von effektiver Zeit ab

Man betrachte den Anfang t_{si} und das Ende t_{ei} des entsprechenden Zeitintervalls i und berechne den Anfangswert der effektiven Zeit i aus der Differenz t_{ei} - t_{si}. Dann wird das ganze Monitor-FIFO nach Unterbrechungszeiten für den Task i durchsucht. Hierbei wird nach Einträgen gesucht, die ein Zeitintervall definieren, das im Zeitintervall i liegt. Wenn ein solches Zeitintervall j gefunden wird, wird dessen Länge von dem aktuellen Wert der effektiven Zeit abgezogen. Am Ende dieses Algorithmus ist der Wert der effektiven Zeit in Prozessorzyklen ermittelt. Er lässt sich anschließend durch eine einfache Umrechnung in Mikrosekunden umwandeln.

7.8 Einfluss des Monitorings auf die Performanz des Systems

Das Monitoring wurde so konzipiert, dass es nur einen minimalen Einfluss auf die Systemperformanz hat. Um diesen Einfluss zu ermitteln, wurde die Performanz der Kommunikationsmechanismen der lokalen Middleware gemessen. Für die Token-basierte Kommunikation konnte keine Performanzeinbuße wegen Monitorings festgestellt werden. Die Zeitdauer des Token-Versands durch den MIRPA-X Scheduler blieb unverändert. Für die synchronen REQUEST-Nachrichten und die asynchronen COMMAND- Nachrichten wurde die Zeitdauer der Nachrichtenübertragung mit und ohne Monitoring gemessen. Für die COMMAND-Nachricht wurde die Zeit zwischen dem Versand der Nachricht in einer Client-Applikation und dem Empfang derselben in einer Server-Applikation aufgenommen. Bei der REQEST-Nachricht

wurde die *Round Trip* Latenz gemessen. Die Messung wurde für Datengrößen von 50 bis 1000 Bytes durchgeführt. Die Ergebnisse sind in Abbildung 7-9 dargestellt. Bei dem REQUEST-Mechanismus wird eine Performanzeinbuße durch das Monitoring von maximal 0,3 µs registriert. Bei dem COMMAND-Mechanismus beträgt die Einbuße maximal 0,4 µs. Zusammenfassend bleibt die Performanzeinbuße infolge des Monitorings minimal. Das Design und die Implementierung des Monitors erfüllen also die an das Echtzeitsystem gestellten Anforderungen.

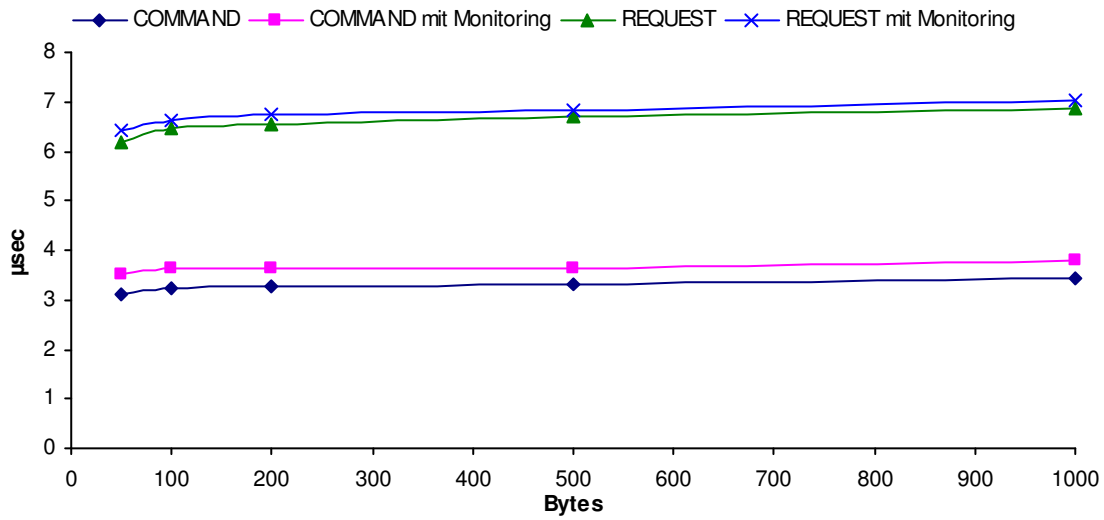

Abbildung 7-9: Einfluss des Monitorings auf die Performanz von COMMAND- und REQUEST-Nachrichten

7.9 System-Monitor im verteilten System

Im verteilten System muss der Monitor in der Lage sein, die Ausführungszeit von entfernten Tasks automatisch zu erfassen und dem Self-Manager zur Verfügung zu stellen. Um dieses Ziel zu erreichen, wird das Monitoring in jedem Netzwerkteilnehmer aktiviert. Auf dem Master-Rechner wird der Monitor-Master gestartet. Eine Slave-Instanz des System-Monitors wird auf jedem Slave-Rechner gestartet. Die Slave-Instanzen sammeln die lokal ermittelten Daten und übermitteln sie an den zentralen Monitor-Master. Die Übermittlung erfolgt über ähnliche passive Mechanismen durch die Datenintegration in den Kommunikationsfluss wie im Abschnitt 7.5 beschrieben. Dadurch wird die Netzlast nicht erhöht und die Echtzeitfähigkeit der verteilten Applikationen nicht durch das Monitoring beeinträchtigt. Die Netzwerkteilnehmer besitzen Systemuhren, die nicht miteinander synchronisiert sind. Außerdem können sie mit unterschiedlichen Taktfrequenzen ausgestattet sein. Aus diesen Gründen können die im Netzwerk von den Monitor-Slaves erfassten Task-Ausführungszeiten nicht mit absoluten Werten an den Monitor-Master verschickt werden. Die Monitor-Slaves übermitteln deshalb

ausschließlich lokal ermittelte effektive Task-Ausführungszeiten an den Monitor-Master. Bis auf die Token-Tasks lassen sich alle Task-Typen auch im verteilten System überwachen.

7.10 Beispielapplikation: der Self-Manager in der Robotersteuerung

Das für die Ausführung der Steuerungssoftware bereitgestellte verteilte System bietet viele Anwendungsmöglichkeiten für das Self-Management. Nach dem in Abschnitt 7.1 eingeführten generischen Entwurfsmodell für Self-Manager-Komponente wurde ein spezieller Self-Manager entwickelt ([MSA+08] [SAG+08] [SGT07] [SGM09]), der eine automatisierte Verteilung von Software-Modulen innerhalb der verteilten Architektur ermöglicht. Im Self-Manager wurden die vier funktionalen Komponenten eines autonomen Managers durch den *QMonitor*, *QAnalizer*, *QPlanner* and *QExecutor* realisiert (Abbildung 7-10). Die funktionale Rolle der QMonitor-Komponente übernimmt der in der Middleware integrierte System-Monitor. Über das shared-memory-basierende Monitor-Interface hat der Self-Manager einen direkten Zugang zu Überwachungsdaten. Durch die Überwachung von System-Topologie und Task-Ausführungszeiten ist er in der Lage, auf Systemänderungen zu reagieren (QAnalizer) und gezielte Verteilungsmuster zu generieren (QPlanner), die gezielt die Ausführung vorhandener Tasks auf die verschiedenen verfügbaren CPU-Ressourcen übertragen (QExecutor).

Das Ziel des Self-Mangers war es, der Steuerungssoftware zu ermöglichen, zusätzliche zur Laufzeit angeschlossene PCs nahtlos zu integrieren und zu nutzen (Selbst-Optimierung), sowie den Ausfall eines PCs durch eine intelligente Rekonfiguration zu kompensieren (Selbst-Heilung). Nach dem Ausfall eines PCs lassen sich die Selbst-Heilungs-Mechanismen allerdings nur unter bestimmten Bedingungen realisieren, wie das folgende Zitat belegt:

„*Zwar wird die Steuerung auf einen solchen Ausfall zunächst mit Einleitung einer Notbremsung reagieren, kann danach aber möglicherweise durch Umsetzung eines anderen Verteilungsmusters den regulären Betrieb wieder aufnehmen. Allerdings muss je nach Task-Konstellation nicht zwangsläufig ein Verteilungsmuster existieren, das den Betrieb mit reduzierter Anzahl von Rechnerressourcen erlaubt [SGM09]*“.

Die resultierende Steuerungssoftware ist in der Lage, sich an veränderte äußere Bedingungen anzupassen und Self-X-Eigenschaften zu realisieren. Je nach festgelegtem Ziel können die generierten Muster das Leistungspotenzial der Steuerungssoftware, z.B. durch höhere Taktfrequenzen, steigern oder eine verbesserte Robustheit durch redundante Task-Ausführung ermöglichen.

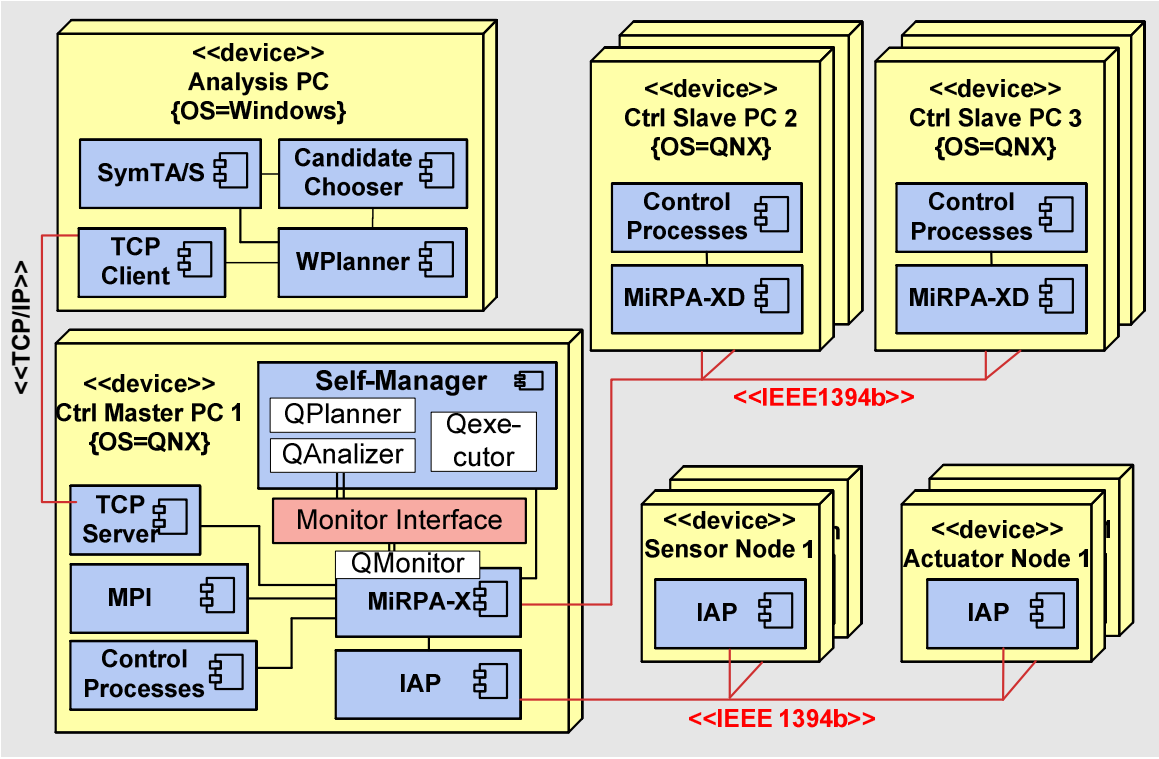

Abbildung 7-10: Self-Manager im verteilten Steuerungssystem nach [SGM09]

7.10.1 Generierung von Verteilungsmustern

Die Generierung von Verteilungsmustern erfolgt mit dem Ziel, eine optimale Nutzung der verteilten CPU-Ressourcen zu erreichen. Im Detail sollen die Netzwerkauslastung sowie die zur einwandfreien Ausführung verteilter Steuerungsaufgaben notwendige Kommunikation zwischen den Rechnern verringert werden. Ferner soll die Gleichverteilung der Prozessorlast auf die verfügbaren CPU-Ressourcen erreicht werden, da diese eine Vorraussetzung für die Erhöhung des Steuerungszyklus ist.

Im verteilten Steuerungssystem erfolgt die Berechnung von Verteilungsmustern auf einem separaten mit Windows ausgestatteten Rechner (Abbildung 7-10), der über eine TCP/IP-Verbindung an den Hauptsteuerungsrechner angebunden ist. Das dabei angewandte Verfahren besteht aus einem Graphpartitionierungsansatz, der in [SAG+08] ausführlicher beschrieben wird. Bei diesem Ansatz werden sämtliche Steuerungskomponenten mit allen Datenabhängigkeiten und Worst-Case-Ausführungszeiten in einen ungerichteten, kanten- und knotengewichteten Graphen durch die Planner-Komponente *WPlaner* (Abbildung 7-10) abgebildet. Dabei steht jeder Knoten für einen Task im System und jede Kante beschreibt die Datenabhängigkeit zwischen zwei Knoten. Jede Kante wird mit der Größe der Daten gewichtet, die zwischen den entsprechenden Knoten ausgetauscht wird. Die Knoten hingegen werden mit den ermittelten Ausführungszeiten der entsprechenden Tasks gewichtet. Durch die Graphenabbildung wird die Berechnung eines geeigneten Verteilungsmusters in eine Graphpartitionierungsaufgabe

umgewandelt, wobei die Anzahl der Partitionen der Anzahl der verfügbaren Rechner entspricht. Ein solch partitionierter Graph ist exemplarisch in Abbildung 7-11 dargestellt. Der angewendete Algorithmus basiert auf dem Verfahren von Kernighan und Lin [KL70]. Sein Ziel ist es, den Graphen so zu partitionieren, dass die Summe der Schnitte minimal bleibt und die Partitionen Tasks enthalten, die in der Summe ausgeglichene Ausführungszeiten aufweisen. Dadurch wird ein Verteilungsmuster erzeugt, das sowohl die Netzwerkkommunikation minimiert als auch für eine ausgewogene Auslastung der verfügbaren Rechner sorgt [SGM09]. Bevor das Verteilungsmuster als brauchbare Lösung eingesetzt werden kann, wird es vom Scheduling-Analysewerkzeug SymTA/S [HHJ+05] auf seine Ausführbarkeit überprüft.

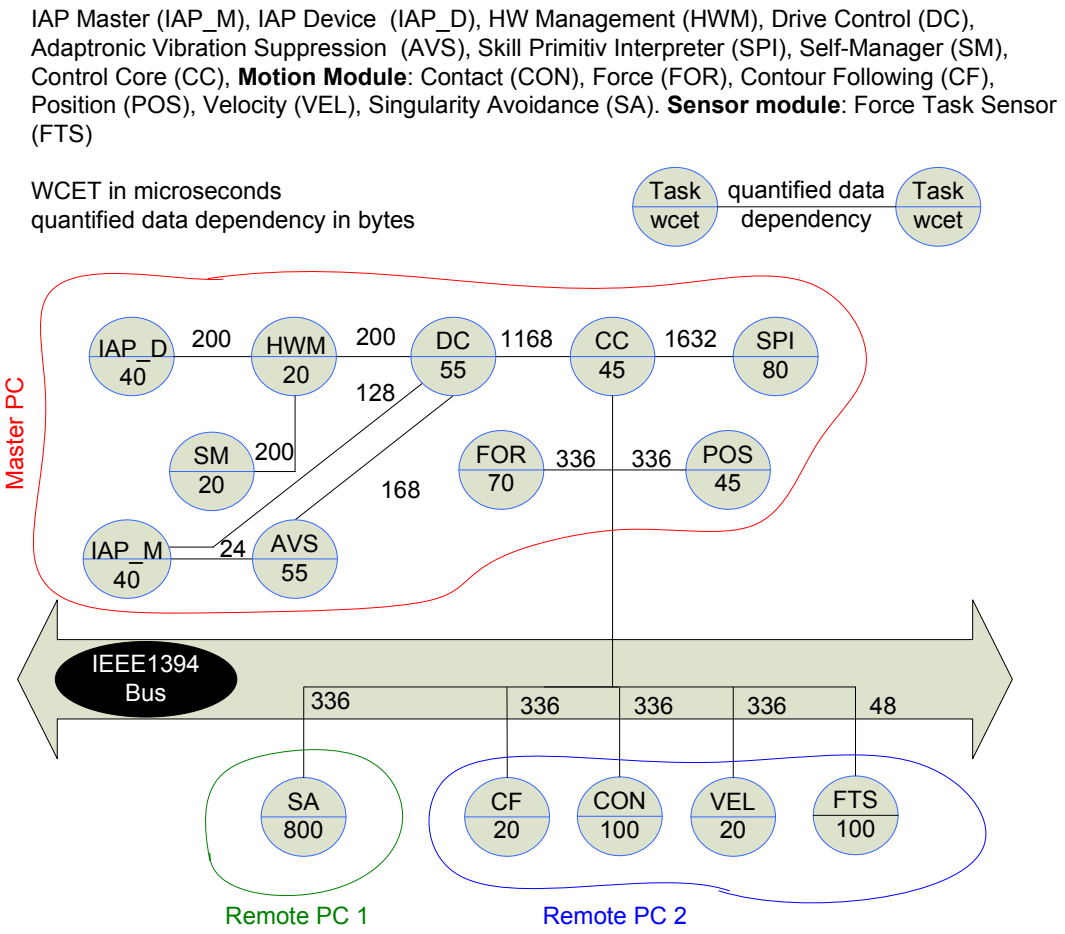

Abbildung 7-11: Darstellung eines anhand des Graphpartitionierungsansatzes berechneten Task-Verteilungsmusters mit drei Rechnerressourcen

7.10.2 Selbst-Heilungs-Mechanismus in der Steuerungssoftware

Wie oben bereits erwähnt ist der Self-Manager bei einer verteilten Ausführung der Steuerungssoftware in der Lage, den Ausfall eines Rechners, auf dem Steuerungssoftware-komponenten ausgeführt werden, durch eine intelligente dynamische Rekonfiguration der Task-

Verteilung zu kompensieren. Der dafür notwendige im Self-Manager integrierte Selbst-Heilungs-Mechanismus ist im Sequenzdiagramm in Abbildung 7-12 dargestellt. Die Ausgangssituation ist folgende: die Steuerungssoftware wird entsprechend eines statisch festgelegten Verteilungsmusters auf drei Rechnern ausgeführt, davon ein Hauptsteuerungsrechner und zwei Remote-Rechner. Die Remote-Rechner sind der Übersicht halber im Sequenzdiagramm nicht eingezeichnet. Neben dem Hauptsteuerungsrechner wird lediglich der auf Windows basierende Analyserechner im Diagramm dargestellt. Auf dem Hauptsteuerungsrechner werden unter anderen die Self-Manager-Komponenten *QMonitor*, *QAnalyzer*, *QPlanner* und *QExecutor* ausgeführt. Auf dem Analyserechner werden die Komponente *WPlanner* und *CandidateChooser* sowie das Scheduling- und Timing-Analysewerkzeug SymTA/S ausgeführt. Bevor Self-X-Funktionalitäten in der Steuerung angewendet werden können, müssen alle Steuerungsmodule (Bewegungs- und Sensormodule), die verteilt werden können, redundant auf sämtliche vorhandene Rechner (lokal und remote) gestartet werden. Die Module werden aber nicht zur Ausführung deren Funktionalitäten aktiviert. Die Aktivierung jener Module, die im zyklischen Betrieb tatsächlich einen Beitrag zur Steuerungsapplikation leisten, erfolgt erst anhand des aktuellen Verteilungsmusters durch den QExecutor.

Zu einem Zeitpunkt t wird ein Remote-Rechner, auf dem Steuerungskomponenten ausgeführt werden, aus dem Netzwerk physikalisch entfernt und dadurch dessen funktionaler Ausfall erzwungen. Durch das Entfernen eines Busteilnehmers generiert der IEEE1394-Bus automatisch einen Bus-Reset, infolge dessen der Bus reinitialisiert wird. Nach der Reinitialisierung werden die neuen Topologieinformationen im Hauptsteuerungsrechner durch die Middleware XD gesammelt. Die Steuerung wechselt daraufhin in einen Fehlerzustand und bleibt in diesem Zustand, bis das Selbst-Heilungsszenario abgeschlossen ist. Angetriggert durch den Bus-Reset schreibt der *QMonitor* die neuen Topologiedaten in das Monitor-Interface und aktiviert anschließend den *QAnalizer*. Letzterer kann anhand einer geeigneten Hash-Funktion die Änderungen im Monitor-Interface schnell erfassen und auswerten. In diesem Fall detektiert er dass Fehlen des Remote-Rechners und leitet den Selbst-Heilungs-Mechanismus ein. Dafür schreibt er die Auswertungsergebnisse in einen vorgesehenen Share-Memory-Bereich (setData(HEAL) aus Abbildung 7-12) und benachrichtigt den *QPlanner*. Der *QPlanner* sucht anschließend in seiner internen Datenbank nach einem gültigen Verteilungsmuster, das bei dem aktuellen Ressourcenbestand angewendet werden kann und stellt es dem *QExecutor* bereit. Falls er kein Verteilungsmuster finden kann, triggert er die Komponente *WPlanner* und *CanditateChooser* auf dem Analyserechner mittels einer TCP/IP-Verbindung an, die ein entsprechendes Muster berechnen und mit SymTA/S validieren.

Zurzeit werden die Implementierungsarbeiten abgeschlossen und erste Tests der Selbst-Heilungs-Eigenschaft werden an der verteilten Steuerungssoftware durchgeführt.

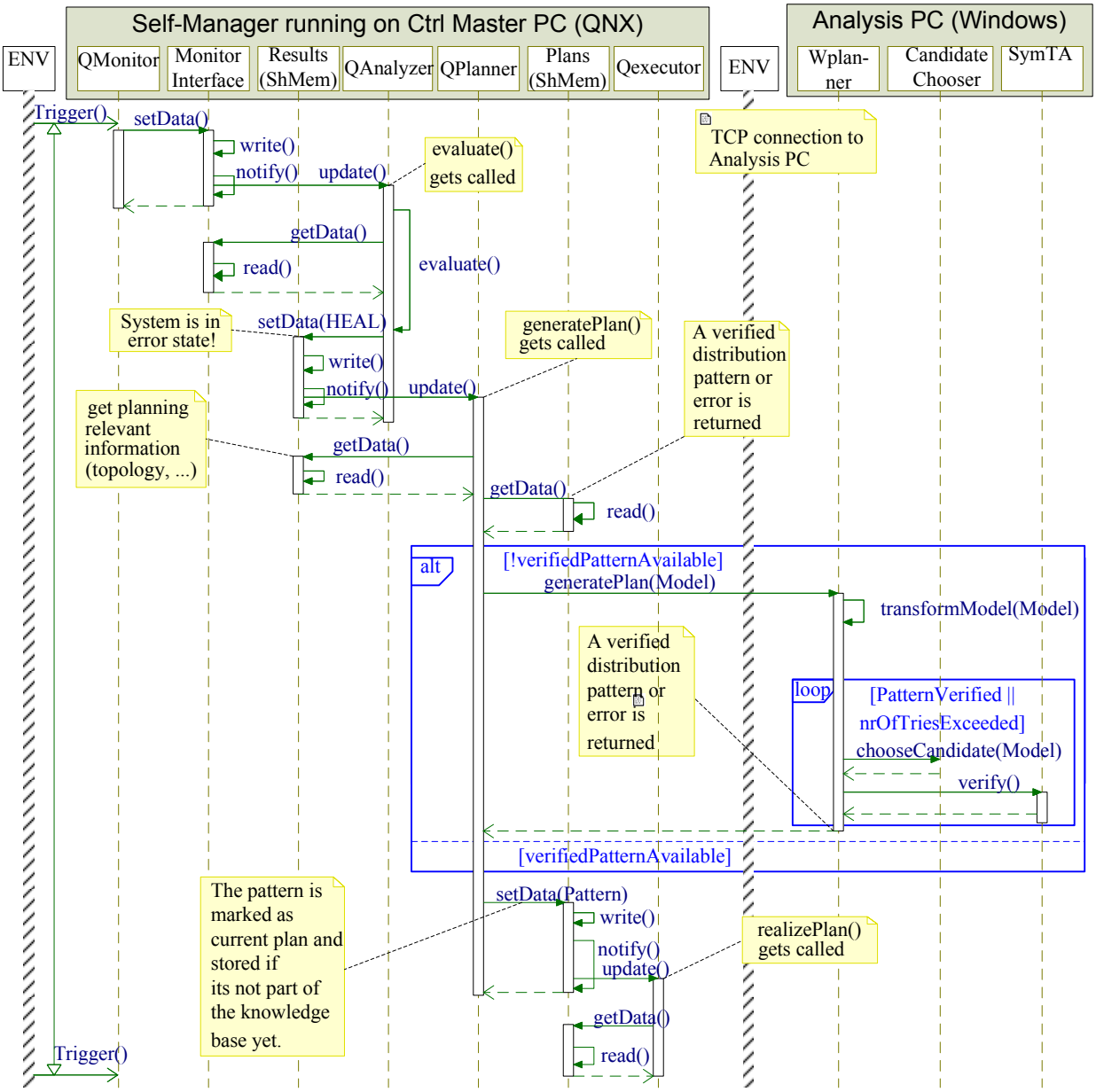

Abbildung 7-12: Selbst-Heilungsszenario nach [SAG+08]

7.10.3 Selbst-Optimierungs-Mechanismus in der Steuerungssoftware

Ein Selbst-Optimierungs-Mechanismus lässt sich ebenso anhand der Abbildung 7-12 beschreiben. Dabei ist die Ausgangssituation folgende: Sämtliche Steuerungskomponenten werden anfänglich auf einem einzigen Rechner ausgeführt. Im laufenden Betrieb werden anschließend zwei beispielshaft vorkonfigurierte Remote-Rechner dazugeschaltet. Die Zuschaltung der Remote-Rechner erzwingt ebenfalls die Generierung eines Bus-Resets gefolgt von der Reinitialisierung des IEEE1394-Busses. Angetriggert durch den Bus-Reset aktualisiert der *QMonitor* die Topologiedaten, inklusiv gesammelter Informationen über die zwei neuen Rechner, in das Monitor-Interface und aktiviert den *QAnalizer*. Letzterer wertet das Monitor-

Interface aus, stellt die hinzugefügten Rechnerressourcen fest und leitet den Selbst-Optimierungs-Mechanismus ein. Anhand der Auswertungsergebnisse sucht der *QPlanner* anschließend in seiner Datenbank nach einem passenden Verteilungsmuster, das sämtliche auszuführende Tasks und die drei Rechnerressourcen berücksichtigt und stellt es dem *QExecutor* zur Ausführung bereit. Falls kein passendes Verteilungsmuster gefunden wird, dann wird ein neues über den WPlanner (7.10.2) berechnet.

Zurzeit ist die Implementierung des Selbst-Optimierungs-Mechanismus weit fortgeschritten und die Validierung an konkreten Szenarien läuf gerade an.

7.11 Zusammenfassung

Zusammenfassend lassen sich, aufbauend auf die in die Middleware MiRPA-XD integrierte Monitoring-Komponente, Self-X-Eigenschaften in die Steuerungsarchitektur RCA562 einbauen. Die Integration von Self-X-Eigenschaften erfolgt anhand einer eng an dem Autonomic-Manager von IBM [KC03]angelehnten Managementkomponente. Die Managementkomponente verleiht der Steuerungssoftware die Möglichkeit, auf dynamische Änderungen der Verfügbarkeit von verteilten Rechenressourcen und sich ändernde aufgabenorientierte Steuerungsanforderungen zu reagieren.

Da die Steuerungssoftwarekomponenten über nachrichtenbasierte Mechanismen der Middleware kommunizieren, lässt sich die für das autonome Computing notwendige Monitoring-Komponente durch Beobachtung und Auswertung des Nachrichtenverkehrs innerhalb der Middleware realisieren. Neben der Generierung von topologischen Informationen ist die Monitoring-Komponente in der Lage, die für das Self-Management notwendige Ausführungszeit einzelner Steuerungs-Tasks durch die Integration von Messpunkten in das MiRPA-XD-Kommunikationsinterface „on the fly“ zu ermitteln. Die Monitoring-Komponente wurde so entworfen, dass die Monitoring-Aktivität keine Auswirkungen auf das Echtzeitverhalten des Systems hat und dessen Performanz nicht oder nur minimal beeinträchtigt. Zurzeit werden die Implementierungsarbeiten an der Integration des Self-Managers in die Steuerung abgeschlossen. Ebenso werden erste Selbst-Heilungs- und Selbst-Optimierungsszenarien in Rahmen eines anderen SFB562-Teilprojekts durchgeführt.

8 Zusammenfassung und Ausblick

Diese Arbeit hatte das Ziel, einen Beitrag zur Steigerung der Performanz und Zuverlässigkeit offener PC-basierter Steuerungssysteme insbesondere im Bereich der Parallelrobotik zu liefern. Aufbauend auf die von Kohn [Koh07] entwickelte Kommunikations-Infrastruktur für hochdynamische Parallelroboter sollte das System so optimiert und erweitert werden, dass Steuerungszyklen mit Frequenzen höher als die bisherigen 1 kHz realisierbar werden, damit eine bessere Ausnutzung des strukturbedingten Potenzials von Parallelrobotern durch höhere Verfahrgeschwindigkeiten und Beschleunigungen ermöglicht wird. Die Erweiterung der Kommunikations-Infrastruktur zu einem verteilten System sollte die verfügbare Rechenleistung skalierbar machen. In diesem Zusammenhang sollte ein geeignetes echtzeitfähiges Bussysem zum Aufbau des Rechennetzwerks eingesetzt werden, damit verteilte echtzeitfähige Kommunikationsmechanismen mit kurzer Latenzzeit realisiert werden können. Weiterhin sollte als Beitrag zur Integration einer Self-Management-Komponente in die Steuerung eine Monitoring-Komponente in die verteilte Middleware integriert werden, die topologische und zeitliche Informationen über die Ausführung von Tasks im verteilten System zur Laufzeit ermittelt und dem Anwender (Self-Manager) zur Verfügung stellt.

Zusammenfassung

Eine Untersuchung der Kommunikations-Infrastruktur ergab, dass mehrere ihrer zugehörigen Komponenten und bereitgestellten Funktionalitäten ein strukturbedingtes Optimierungspotenzial hinsichtlich der erreichbaren Systemperformanz aufweisen. Im Einzelnen wurden die Softwarekomponenten MiRPA-X und IAP sowie das Hardware-Design und die Datenverarbeitung innerhalb der Kommunikationsmodule untersucht.

In der zentralen Softwarekomponente MiRPA-X konnten Funtionalitäten und Entwurfsmuster identifiziert werden, die einen hierarchischen und auf Performanz optimierten Aufbau der Steuerungsarchitektur erschweren. Dazu zählt das Client/Server-Modell, das so geändert wurde, dass Applikationsprozesse gleichzeitig als Client und Server agieren dürfen. Außerdem wurde die synchrone Nachrichtenkommunikation so erweitert, dass umfangreiche Applikationsdaten mit einer REQUEST-Anfrage an einen Server übermittelt werden können und auf diese Weise performanzkostende Desing-Konstrukte in der Steuerungssoftware vermieden werden. Ferner wurde das Konfliktpotenzial bei der Abarbeitung von Konfigurations- und IPC-Nachrichten in der Middleware durch eine multi-threaded Erweiterung der Middleware gelöst, bei der beide Nachrichtenklassen parallel von zwei unterschiedlichen Threads abgearbeitet werden. Ferner konnte durch die Erweiterung des Token-Schedulings, die die parallele Ausführung von Token-Threads ohne Datenabhängigkeit auf Mehrkernprozessoren ermöglicht, die auf diesen Architekturen verfügbare Rechenleistung besser ausgenutzt werden.

In das IAP-Protokoll wurde zur Erhöhung der Betriebsicherheit ein fehlertoleranter Mechanismus in die Fehlerbehandlung integriert, der die zyklische Kommunikation robuster macht gegen gelegentlich auftretende Telegrammfehler.

Eine Analyse des Hardwaredesigns und der Software-Abläufe auf den Kommunikationsknoten hat gezeigt, dass letztere sowohl in Hard- als auch in Software ein großes Optimierungspotenzial aufweisen. Aus den gewonnenen Erkenntnissen konnte ein neuer Hardware-Aufbau für die Kommunikationsmodule entworfen werden, bei dem entscheidende und zur Datenverarbeitung beitragende Komponenten in ein FPGA auf Chip-Level integriert wurden. Durch die Hardware-Integration konnten die Datenzugriffszeiten reduziert und mehrere Kopieroperationen, Synchronisations- und Totzeiten innerhalb des Datenverarbeitungsprozess vermieden werden. Als Ergebnis konnte die zyklische Datenverarbeitungszeit, die innerhalb eines Kommunikationsmoduls zwischen dem Eingang eines MDT und dem Versand des nächsten DDT gemessen wird, um den Faktor 32 von 480 µs auf 15 µs verbessert werden. Dadurch konnte der zeitliche Anteil der Kommunikation im zyklischen Betrieb von ca. 640 µs auf 60 µs reduziert werden. Dadurch wurde die Grundlage für die Umsetzung eines Steuerungszyklus mit einer Frequenz bis theoretisch 8 kHz gelegt. In der Praxis wurde an SFB-Demonstratoren ein Steuerungszyklus mit einer maximalen Frequenz von 2 kHz umgesetzt. Die Begrenzung der Steuerungsfrequenz ist in dem Fall durch die eingesetzten Frequenzumrichter bedingt.

Durch die Erweiterung der Middleware auf der verteilten Version MiRPA-XD wurde eine Möglichkeit dafür geschaffen, die Berechnung aufwendiger Algorithmen im zyklischen Betrieb auf verteilten Rechnern auszuführen. Anstatt wie bisher einen einzigen Rechner in die PC-basierte Steuerung einzusetzen, kann unter XD ein Rechnernetzwerk eingesetzt werden. In dem Rechnernetzwerk realisiert XD ein Master/Slaver-Verteilungsmodell; auf dem Master-Rechner bzw. auf dem Slave-Rechner wird eine XD-Master-Instanz bzw. eine XD-Slave-Instanz ausgeführt. Der bisherige Steuerungsrechner fungiert dabei als Master-Rechner. Aus Echtzeit- und Performanzgründen wurde, neben der vom IAP geregelten zyklischen Kommunikation, eine zweite physikalische auf IEEE1394 Standard basierende Verbindung zwischen Master und Slaves realisiert. XD unterstützt die dynamische Integration neuer Slaves im laufenden Betrieb. Unter XD erfolgt der Zugriff auf entfernte Ressourcen über den auf Netzwerkebene ausgeweiteten *Message-Passing* Mechanismus. Dabei macht XD den Einsatz der bisherigen nachrichtenbasierten COMMAND- und REQUEST-Mechanismen im verteilten System für den Anwender transparent. Durch den Einsatz des IEEE1394 Standards ist der Zugriff auf verteilte Ressourcen unter XD deterministisch und erfüllt somit eine grundsätzliche Bedingung für die verteilte Ausführung von Steuerungssoftwarekomponenten. Dank des einfachen Designs und der konsequenten Implementierung weist XD im Vergleich mit anderen verteilten Systemen eine bessere Performanz beim Zugriff auf verteilten Ressourcen auf.

Durch die Integration einer Monitoring-Komponente in die Middleware MIRPA-XD konnte die Grundlage für die Integration von Self-X-Eigenschaften in die Steuerungsarchitektur RCA562 gelegt werden. Die darauf aufbauende Managementkomponente ermöglicht es der Steuerungsoftware, dynamisch auf unvorhergesehene Ereignisse zu reagieren und eine aufgabenorientierte und leistungsoptimierte Nutzung der verfügbaren verteilten Ressourcen zu realisieren. Die Rolle der Monitoring-Komponente umfasst die Generierung von topologischen Informationen und die Ermittlung der Ausführungszeit einzelner Steuerungs-Tasks im laufenden

Betrieb. Die ausgeführten Monitoring-Aktivitäten haben keine Auswirkung auf das Echtzeitverhalten und auf die Performanz des Systems. Zum jetzigen Zeitpunkt werden Implementierungsarbeiten an der Managment-Komponente abgeschlossen und erste Selbst-Heilungs- und Selbst-Optimierungsszenarien in die Steuerung integriert.

Zusammenfassend leistet diese Arbeit einen Beitrag zur Steigerung der Performanz und Zuverlässigkeit offener PC-basierter Steuerungssysteme für Parallelroboter.

Ausblick

Die während dieser Arbeit im Rahmen des Sonderforschungsbereichs 562 entwickelte verteilte Kommunikationsarchitektur basiert auf einem Master/Slave-Modell. Im diesem Modell wird der Master-Rechner als zentraler Steuerungsrechner eingesetzt. Trotz der verteilten Struktur ist ein theoretischer Ausfall des Master-Rechners derzeit nicht zu kompensieren und bedeutet einen Ausfall der Steuerungsapplikation. Um Steuerungsapplikationen im verteilten Kontext robuster gegenüber einem möglichen Ausfall des Master-Rechners zu machen, kann in Zukunft, aufbauend auf die Ergebnisse dieser Abeit, eine redundante Auslegung der Steuerung auf sämtliche verfügbare Rechner realisiert werden. Bei der redundanten Steuerung soll ein Slave-Rechner die Master-Funtionalität übernehmen und die Steuerungsaktivitäten weiterführen, falls der Master-Rechner ausfällt. Dafür ist es notwendig, dass Daten und Statusinformationen der Steuerung zwischen den verfügbaren Rechnern so synchronisiert werden, dass jeder Rechner stets den aktuellen Zustand der Steuerungsapplikation kennt. Um das Volumen des durch Synchronisation entstehenden Datenverkehrs zu reduzieren, ist eine Erweiterung der Netzwerkstruktur nach Abbildung 8-1 erforderlich. Dort werden die Slave-Rechner über eine zusätzliche physikalische Verbindung mit den Sensor- und Aktoreneinheiten verbunden. Aufbauend auf dieser zusätzlichen physikalischen Verbindung kann der Datenaustausch im IAP so erweitert werden, dass Sensordaten (DDT) im zyklischen Betrieb sowohl an den Master-Rechner als auch an alle verfügbaren Slave-Rechner gleichzeitig versendet werden. Auf diese Weise wird, ohne zusätzlichen Datenaustausch zwischen Master- und Slave-Rechner, dafür gesorgt, dass in jedem Zyklus die aktuellen Sensordaten immer auf jedem Rechner vorhanden sind.

Außerdem ermöglicht der gleichzeitige Versand von Sensordaten an alle Rechner die Realisierung des in [DMM+08] eingeführten „pre-request“-Ausführungsansatzs. Der Begriff „pre-request“ bezieht sich auf das Request/Reply-Kommunikationsmodell, bei dem verteilte Tasks mittels Request-Nachrichten zur Ausführung aktiviert werden. Da Sensordaten gleichzeitig auf jedem Rechner aktualisiert werden, kann die Sensordatenverarbeitung auf verteilte Rechner unmittelbar nach Eingang des Sensordaten-Events getriggert und parallel zur Task-Ausführung auf dem Master-Rechner ausgeführt werden. Der Algorithmus zur Sensordatenverarbeitung wird also ausgeführt, bevor eine Applikation auf dem Master-Rechner explizit durch eine Request-Nachricht den Auftrag zur Ausführung gibt. Ein Performanzvorteil lässt sich dadurch erzielen, dass bei einer späteren Request-Anfrage die Ergebnisse der

Ausführung auf dem verteilten Rechner bereits zur Verfügung stehen und nicht mehr berechnet werden müssen. Die Antwort auf die Anfrage kann dementsprechend im besten Fall ohne weitere Verzögerung zurück zum Master-Rechner geschickt werden.

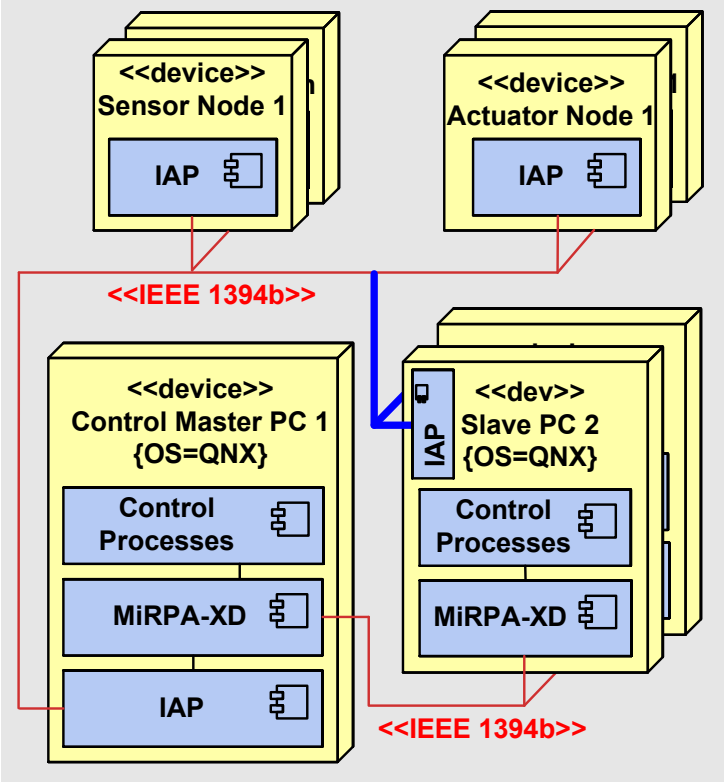

Abbildung 8-1: Erweiterung der Netzwerkstruktur der Kommunikations-Infrastruktur mit zusätzlichen physikalischen Verbindungen zwischen Slave-Rechnern und Sensor/Aktoreinheiten

Abkürzungen

CST: Cycle Start Telegram

DDT: Device Data Telegram

DPRAM: Dual Port Random Access Memory

GUI: Graphical User Interface

IAP: Industrial Automation Protocol

IC: Integrated Circuit

KK: Kommunikationsknoten

LLC: Link Layer Controller

MDT: Master Data Telegram

MiRPA-XD: M*i*ddleware for Robotic and Process Control Application

PHY: Physical Layer

RCA562: Robot Control Architecture 562

SFB562: Sonderforschungsbereich 562

XML: eXtensible Markup Language

Abbildungsverzeichnis

Literaturverzeichnis

[And99] D. Anderson: *FireWire System Architecture.* Addison Wesley, 1999.

[Bec01] G. Beckmann: *Ein Hochgeschwindigkeits-Kommunikations-System für die industrielle Automation.* Dissertation, Braunschweig, 2001.

[BEC10] BECKHOFF Automation: *SPS und Motion Control auf dem PC.* http://www.beckhoff.de/default.asp?twincat/default.htm, 04.2010.

[BH06] B. Bäuml and G. Hirzinger: *Agile robot development (aRD): A pragmatic approach to robotic software:* Proceedings of the IEEE/RSJ International Conference on Intelligent Robots and Systems, Beijing, 2006.

[BKM+05] A. Brooks ,T. Kaupp ,A. Makarenko ,S. Williams and A. Oreback: *Towards component-based robotics:* Proceedings of the IEEE/RSJ International Conference on Intelligent Robots and Systems, Edmonton, Canada, 2005; p. 163–168.

[BMC+05] C. Bier ,J. Maaß ,A. Campos and E. Queiroz: *Direct Singularity Avoidance Strategy for the HEXA Parallel Robot:* 18th International Congress of Mechanical Engineering (COBEM 2005), Ouro Preto, Brasilien, 2005.

[BS04] J. Baeten and J. de Schutter: *Integrated visual servoing and force control.* http://www.zentralblatt-math.org/zmath/en/search/?an=1033.93001.

[CLM+04] C. Cote ,D. Letourneau ,F. Michaud ,J. M. Valin ,Y. Brosseau ,C. Raievsky ,M. Lemay and V. Tran: *Code reusability tools for programming mobile robots:* Proceedings of the IEEE/RSJ International Conference on Intelligent Robots and Systems, Sendai, Japan, 2, 2004; p. 1820-1825 vol.2.

[CLZ+98] N. Costescu ,M. Loffler ,E. Zergeroglu and D. Dawson: *QRobot - a multitasking PC based robot control system:* Proceedings of the 1998 IEEE International Conference onControl Applications, 1998; p. 892–896.

[Cor07] Cor: *FireLink Basic Manual, 1394b Link Layer Controller IP Core,* 16.12.2009.

[Cor12] Corba tutotials: http://www.cs.wustl.edu/*schmidt/tutorials-corba.html, 04.2012

[CYP97] CYPRESS: *Datasheet, 1K x 8 Dual-Port Static RAM,* 04.01.2010.

[Dap09] DapTechnology. http://www.daptechnology.com/, 16.12.2009.

[DMK+10] Y. Dadji ,H. Michalik ,N. Kohn ,J. Steiner ,G. Beckmann ,T. Möglich and J. U. Varchmin: *A Communication Architecture for Distributed Real-Time Robot Control:* Robotic Systems for Handling and Assembly. Springer, 2010.

[DMM+07] Y. Dadji ,H. Michalik ,T. Möglich ,N. Kohn and J. Steiner: *Performance Optimized Communication System for High-Dynamic and Real-Time Robot Control Systems:* Proceedings of 16th International Workshop on Robotics in Alpe-Adria-Danube Region (RAAD 2007), 2007; p. 192–201.

[DMM+08] Y. Dadji ,J. Maaß ,H. Michalik ,T. K. N. Möglich and J.-U. Varchmin: *Networked Architecture for Distributed PC-based Robot Control*

Systems: proceedings of the International Conference on Automation, Robotics and Control, Orlando (FL), USA, 2008.

[DMM08] Y. Dadji ,J. Maaß and H. Michalik: *Parallel Task Processing on a Multicore Platform in a PC-based Control System for Parallel Kinematics:* Proceedings of the 6th International Conference on Computing, Communications and Control Technologies (CCCT), Orlando (FL), USA, 2008.

[DMM09] Y. Dadji ,H. Michalik and T. Möglich: *Parallel Architecture for Real Time Control System Based on the IEEE1394 Standard:* Proceedings of the International Conference on Parallel and Distributed Processing Techniques and Applications, Las Vegas (NV), USA, 2009.

[EK01] U. Eberhardt and H. J. Kelm: *USB 2.0.* Datendienste, Function, Hub, Host, Errorhandling, Powermanagement, USB-Treiber, USB-Bausteine, USB-Applikationen, Test & Analyse. Franzis, Poing, 2001.

[Eth03] Ethernet powerlink standardization group: *Ethernet powerlink communication profile specification v. 2.0,* 2003.

[FCI10] *Fibre Channel Industry Association.* http://www.fibrechannel.org/, 19.07.2010.

[Fin04] B. Finkemeyer: *Robotersteuerungsarchitektur auf der Basis von Aktionsprimitiven.* Techn. Univ., Diss.--Braunschweig. Shaker, Aachen, 2004.

[Fis02] S. Fischer: *Vorlesung "verteilte Systeme"*, http://www.ibr.cs.tu-bs.de/courses/ws0203/vs/PDF/VS-0203-Kap01-Einfuehrung-4S.pdf

[FKK+07] B. Finkemeyer ,T. Kröger ,D. Kubus ,M. Olschewski and F. M. Wahl: *MiRPA: Middleware for Robotic and Process Control Applications:* Proceedings of the IEEE/RSJ International Conference on Intelligent Robots and Systems, San Diego, USA, 2007.

[FKW10] B. Finkemeyer ,T. Kröger and F. M. Wahl: *A Middleware for High-Speed Distributed Real-Time Communication:* Robotic Systems for Handling and Assembly. Springer, 2010.

[Gee01] D. Gee: *The how's and why's of PC based control:* Pulp and Paper Industry Technical Conference, 2001 Conference Record of, 2001; p. 67–74.

[HAH09] S. Hassan ,D. Al-Jumeily and A. J. Hussain: *Autonomic Computing Paradigm to Support System's Development:* Second International Conference on Developments in eSystems Engineering (DESE), 2009; p. 273–278.

[HHC96] S. Hutchinson ,G. D. Hager and P. I. Corke: *A tutorial on visual servo control. In* IEEE Transactions on Robotics and Automation, 1996, 12; p. 651–670.

[HHJ+05] R. Henia ,A. Hamann ,M. Jersak ,R. Racu ,K. Richter and R. Ernst: *System level performance analysis - the SymTA/S approach. In* IEE Proceedings Computers and Digital Techniques, 2005, 152; p. 148–166.

[HMB05] J. Hesselbach ,J. Maaß and C. Bier: *Singularity Prediction for Parallel Robots for Improvement of Sensor-Integrated Assembly. In* CIRP Annals - Manufacturing Technology, 2005, 54; p. 349–352.

[IEE08] IEEE: *Standard for a High Performance Serial Bus, IEEE Std. 1394-2008,* 2008.

[ISO96] ISO/IEC: *Information Technology - Open system Interconnection - Basic Reference Model: The Basic Model,* 1996.

[Kal10] D. Kalinsky: *D. Kalinsky Associates - Whitepaper "Designing Software for Multicore Systems".* http://www.kalinskyassociates.com/Wpaper9.html, 26.02.2010.

[KC03] J. O. Kephart and D. M. Chess: *The vision of autonomic computing. In* Computer, 2003, 36; p. 41–50.

[KL70] B. W. Kernighan and S. Lin: *An Efficient Heuristic Procedure for Partitioning Graphs. In* The Bell system technical journal, 1970, 49; p. 291□307.

[Koh07] N. Kohn: *Kommunikations-Infrastruktur für hochdynamische Parallelroboter.* Dissertation, Braunschweig, 2007.

[KUK10]: *KUKA Roboter Steuerung (KR C).* http://www.kuka-robotics.com/switzerland/de/solutions/solutions_search/?appl_prod=12, 19.04.2010.

[Maa09] J. H. Maaß: *Ein Beitrag zur Steuerungstechnik für parallelkinematische Roboter in der Montage.* Techn. Univ., Diss.--Braunschweig. Vulkan-Verl., Essen, 2009.

[MAC06]: *Macro home.* http://www.macro.org/, 15.01.2010.

[Mas81] M. T. Mason: *Compliance and Force Control for Computer Controlled Manipulators. In* IEEE Transactions on Systems, Man and Cybernetics, 1981, 11; p. 418–432.

[MDM08] H. Michalik ,Y. Dadji and J. Maaß: *A Communication Architecture for Support of Distributed Computing in Real-Time Environments:* Proceedings of 3nd International Colloquium of the Collaborative Research Center 562, Braunschweig, Germany, 2008.

[Mer97] J.-P. Merlet: *Les Robots parallèles.* Edition Hermès, Paris, 1997.

[MHS+07] J. Maaß ,J. Hesselbach ,J. Steiner and U. Goltz: *Self-Management in a Robot Control Architecture:* Proceedings of the Second International Workshop on Software Development and Integration in Robotics (SDIR), affiliated with ICRA, Italy, 2007.

[MKH06] J. Maaß ,N. Kohn and J. Hesselbach: *Open Modular Robot Control Architecture for Assembly Using the Task Frame Formalism. In* International Journal of Advanced Robotic Systems, 2006, 3; p. 1–10.

[MSA+08] J. Maaß ,J. Steiner ,A. Amado ,J. Hesselbach ,M. Huhn and A. Raatz: *Self-Management In A Control Architecture For Parallel Kinematic Robots:* Proceedings of the ASME 2008 International Design Engineering Technical Conferences & Computers and Information in Engineering Conference (IDETC/CIE), Brooklyn, New York, USA, 2008.

[Mun10] H. Munz: *Robotersteuerung meets Motion Control.* http://www.sps-magazin.de/?inc=artikel/article_show&nr=26524, 19.04.2010.

[NAS10]: *CLARAty Robotic Software.* http://claraty.jpl.nasa.gov/man/overview/index.php, 30.06.2010.

[Orc08]: *Orca Robotics.* http://orca-robotics.sourceforge.net/head/hydro/, 01.07.2010.

[Orc09]: *Orca Robotics.* http://orca-robotics.sourceforge.net/orca_doc_performance.html, 19.03.2010.

[ORO09] OROCOS Homepage: *Open robot control software, www.orocos.org,* 2009.

[SOE06] Peter Soetens: *Open robot control software.* In V Jornades de Programari Lliure, Barcelona, 2006 www.orocos.org, 2009.

[pla10]: *Player Project.* http://playerstage.sourceforge.net/, 01.07.2010.

[PM05] R. Pigan and M. Metter: *Automatisieren mit PROFINET - Industrielle Kommunikation auf Basis von Ethernet.* Siemens AG, 2005.

[Pry08] G. Prytz: *A performance analysis of EtherCAT and PROFINET IRT:* Emerging Technologies and Factory Automation, 2008. ETFA 2008. IEEE International Conference on, 2008; p. 408–415.

[PW97] P. Lutz and W. Sperling: *OSACA, the vendor neutral control architecture:* Proceedings of the Iim‘97, Internatioanl Conference, 1997.

[QNX09] QNX: *The Instrumented Microkernel.* http://www.qnx.com/developers/docs/6.4.1/neutrino/sys_arch/trace.html, 18.11.2009.

[RJ98] D. H. R. Orfali and J. Edwards: Instant CORBA, 1998.

[SAG+08] J. Steiner ,A. Amado ,U. Goltz ,M. Hagner and M. Huhn: *Engineering Self-Management into a Robot Control System:* Proceedings of 3nd International Colloquium of the Collaborative Research Center 562, Braunschweig, Germany, 2008; p. 113–125.

[SER07]: *SERCOS interface - Competence in Motion.* http://www.sercos.org/, 19.07.2010.

[SFB10] SFB 562: *Robotersysteme für Handhabung und Montage.* http://www.tu-braunschweig.de/sfb562/index.html, 30.04.2010.

[SGM09] J. Steiner ,U. Goltz and J. Maaß: *Dynamische Verteilung von Steuerungskomponenten unter Erhalt von Echtzeiteigenschaften:* 6. Paderborner Workshop Entwurf mechatronischer Systeme, 2009.

[SGT07] J. Steiner ,U. Goltz and H. Tianfield: *Engineering Self-Management into Legacy Systems. In* System and Information Science Notes, 2007, 2; p. 114–117.

[SH07] J. Steiner and M. Hagner: *Runtime Analysis of a Self-Adaptive Hard Real-Time Robotic Control System:* Proceedings of 4th IEEE Workshop on Engineering of Autonomic and Autonomous Systems (EASe 2007). IEEE Computer Society, Tucson, Arizona, USA, 2007; p. 53–60.

[SHG07] J. Steiner ,M. Hagner and U. Goltz: *Runtime Analysis and Adaptation of a Hard Real-Time Robotic Control System. In* Journal of Computers (JCP), 2007, 2; p. 18–27.

[SMR+04] Stephan Algermissen ,Michael Rose ,Ralf Keimer ,Elmar Breitbach and Eric H. Anderson: *High-speed parallel robots with integrated vibration suppression*

for handling and assembly. In Smart Structures and Materials 2004: Industrial and Commercial Applications of Smart Structures Technologies, 2004, 5388; p. 1–10.

[Sv88] J. de Schutter and H. van Brussel: *Compliant Robot Motion I. A Formalism for Specifying Compliant Motion Tasks. In* The International Journal of Robotics Research, 1988, 7; p. 3–17.

[SVZ09] L. Seno ,S. Vitturi and C. Zunino: *Real Time Ethernet networks evaluation using performance indicators:* Emerging Technologies & Factory Automation, 2009; p. 1–8.

[TMH+05] U. Thomas ,J. Maaß ,J. Hesselbach and F. M. Wahl: *Towards a New Concept of Robot Programming in High Speed Assembly Applications:* Proceedings of the IEEE/RSJ International Conference on Intelligent Robots and Systems, Edmonton, Canada, 2005; p. 3932–3938.

[USE+02] H. Utz ,S. Sablatnog ,S. Enderle and G. Kraetzschmar: *Miro - middleware for mobile robot applications. In* IEEE Transactions on Robotics and Automation, 2002, 18; p. 493–497.

[VDM10] VDMA: *Robotik, Forschung und Innovation.* http://www.vdma.org, 23.04.2010.

[Ver09] D. C. Verma: *Principles of Computer Systems and Network Management.* Springer, 2009.

[VGH03] R. T. Vaughan ,B. P. Gerkey and A. Howard: *On device abstractions for portable, reusable robot code:* Proceedings of the IEEE/RSJ International Conference on Intelligent Robots and Systems, 2003; p. 2421-2427 vol.3.

[Xil03]: *PowerPc 405 Processor Block Reference Guide,* 05.01.2010.

[Xil08]: *MicroBlaze Processor Reference Guide,* 05.01.2010.

[Xil99] Xilinx: *Platform Flash In-System Programmable Configuration PROMs, Data Sheet.* http://www.xilinx.com/support/documentation/data_sheets/ds123.pdf, 13.07.2010.

[Zer10]: *The Home of Ice.* http://www.zeroc.com/index.html, 01.07.2010.